Algorithmica (2016) 74:1–48
DOI 10.1007/s00453-014-9931-3

One-Variable Word Equations in Linear Time

Artur Jeż

Received: 22 January 2014 / Accepted: 7 August 2014 / Published online: 11 September 2014

Abstract In this paper we consider word equations with one-variable (and arbitrarily many occurrences of it). A recent technique of recompression, which is applicable to general word equations, is shown to be suitable also in this case. While in general case the recompression is nondeterministic in case of one-variable it becomes deterministic and its running time is $\mathcal{O}(n + \#_X \log n)$, where $\#_X$ is the number of occurrences of the variable in the equation. This matches the previously best algorithm due to Dąbrowski and Plandowski. Then, using a couple of heuristics as well as more detailed time analysis, the running time is lowered to $\mathcal{O}(n)$ in the RAM model. Unfortunately, no new properties of solutions are shown.

Keywords Word equations · String unification · One-variable equations

1 Introduction

1.1 Word Equations

The problem of satisfiability of word equations was considered as one of the most intriguing in computer science and its study was initiated by Markow already in the 50s. The first algorithm for it was given by Makanin [15], despite earlier conjectures that the problem is undecidable. The proposed solution was very complicated in terms

A. Jeż
Max Planck Institute fü Informatik, 66123 Campus E1 4, Saarbrücken, Germany
e-mail: ajez@mpi-inf.mpg.de

A. Jeż (✉)
Institute of Computer Science, University of Wrocław, ul. Joliot-Curie 15, 50-383 Wrocław, Poland
e-mail: aje@cs.uni.wroc.pl

of proof-length, algorithm and computational complexity. It was improved several times, however, no essentially different approach was proposed for over two decades.

An alternative algorithm was proposed by Plandowski and Rytter [21], who showed that each minimal solution of a word equation is exponentially compressible, in the sense that for a word equation of size n and minimal solution of size N the LZ77 (a popular practical standard of compression) representation of the minimal solution is polynomial in n and $\log N$. Hence a simple non-deterministic algorithm that guesses a compressed representation of a solution and verifies the guess has running time polynomial in n and $\log N$. However, at that time the only bound on N followed from Makanin's work (with further improvements) and it was triply exponential in n.

Soon after Plandowski showed, using novel factorisations, that N is at most doubly exponential [18], showing that satisfiability of word equations is in NEXPTIME. Exploiting the interplay between factorisations and compression he improved the algorithm so that it worked in PSPACE [19].

Producing a description of all solutions of a word equation, even when a procedure for verification of its satisfiability is known, proved to be also a non-trivial task. Still, it is also possible to do this in PSPACE [20], though insight and non-trivial modifications to the earlier procedure are needed.

On the other hand, it is only known that the satisfiability of word equations is NP-hard.

1.1.1 Two Variables

Since in general the problem is outside P, it was investigated, whether some subclass of it is feasible, with a restriction on the number of variables being a natural candidate. It was shown by Charatonik and Pacholski [2] that indeed, when only two variables are allowed (though with arbitrarily many occurrences), the satisfiability can be verified in deterministic polynomial time. The degree of the polynomial was very high, though. This was improved over the years and the best known algorithm is by Dąbrowski and Plandowski [3] and it runs in $\mathcal{O}(n^5)$ and returns a description of all solutions.

1.1.2 One-Variable

Clearly, the case of equations with only one-variable is in P. Constructing a cubic algorithm is almost trivial, small improvements are needed to guarantee a quadratic running time. First non-trivial bound was given by Obono, Goralcik and Maksimenko, who devised an $\mathcal{O}(n \log n)$ algorithm [17]. This was improved by Dąbrowksi and Plandowski [4] to $\mathcal{O}(n + \#_X \log n)$, where $\#_X$ is the number of occurrences of the variable in the equation. Furthermore they showed that there are at most $\mathcal{O}(\log n)$ distinct solutions and at most one infinite family of solutions. Intuitively, the $\mathcal{O}(\#_X \log n)$ summand in the running time comes from the time needed to find and test these $\mathcal{O}(\log n)$ solutions.

This work was not completely model-independent, as it assumed that the alphabet Γ is finite or that it can be identified with numbers. A more general solution was presented by Laine and Plandowski [13], who improved the bound on the number of solutions to $\mathcal{O}(\log \#_X)$ (plus the infinite family) and gave an $\mathcal{O}(n \log \#_X)$ algorithm that

runs in a pointer machine model (i.e. letters can be only compared and no arithmetical operations on them are allowed); roughly one candidate for the solution is found and tested in linear time. Note that there is a conjecture that one-variable word equations have $\mathcal{O}(1)$ solutions (plus the infinite family), in fact, an equation with three solutions outside the infinite family is not known.

1.2 Recompression

Recently, the author proposed a new technique of *recompression* based on previous techniques of Mehlhorn et al. [16] (for dynamic text equality testing), Lohrey and Mathissen [14] (for fully compressed membership problem for NFAs) and Sakamoto [22] (for construction of the smallest grammar for the input text). This method was successfully applied to various problems related to grammar-compressed strings [5,6,8]. Unexpectedly, this approach was also applicable to word equations, in which case alternative proofs of many known algorithmic results were obtained using a unified approach [7]. Recently, it was also extended from strings to trees [10], in particular the algorithm for word equations was generalised to context unification [9].

The technique is based on iterative application of two replacement schemes performed on the text t:

- *pair compression of* ab: For two different letters a, b such that substring ab occurs in t replace each of ab in t by a fresh letter c.
- *a's block compression:* For each maximal block a^ℓ, where a is a letter and $\ell > 1$, that occurs in t, replace all a^ℓs in t by a fresh letter a_ℓ.

In one phase, pair compression (block compression) is applied to all pairs (blocks, respectively) that occurred at the beginning of this phase. Ideally, each letter is then compressed and so the length of t halves, in a worst-case scenario during one phase t is still shortened by a constant factor.

The surprising property is that such a schema can be efficiently applied to grammar-compressed data [5,8] or to text given in an implicit way, i.e. as a solution of a word equation [7]. In order to do so, local changes of the variables (or nonterminals) are needed: X is replaced with $a^\ell X$ (or Xa^ℓ), where a^ℓ is prefix (suffix, respectively) of the substitution for X. In this way the solution that substitutes $a^\ell w$ (or wa^ℓ, respectively) for X is implicitly replaced with one that substitutes w.

1.2.1 Recompression and One-Variable Equations

Clearly, as the recompression approach works for general word equations, it can be applied also to restricted subclasses. However, while in case of word equations it heavily relies on the nondeterminism, when restricted to instances with one-variable it can be easily determinised; Sect. 2 recalls the main notions of word equations and recompression. Furthermore, a fairly natural implementation has $\mathcal{O}(n + \#_X \log n)$ running time, so the same as the Dąbrowski and Plandowski algorithm [4]; this is presented in Sect. 3. Lastly, adding a few heuristics, data structures as well as applying a more sophisticated analysis yields a linear running time, this is described in Sect. 4.

1.3 Outline of the Algorithm

In this paper we present an algorithm for one-variable equations based on the recompression. It also provides a compact description of all solutions of such an equation. Intuitively: when pair compression is applied, say ab is replaced by c (assuming it *can* be applied) then there is a one-to-one correspondence of the solutions before and after the compression, this correspondence is simply an exchange of all abs by cs and vice-versa. The same applies to the block compression. On the other hand, the modification of X can lead to loss of solutions (note that for technical reasons we do not consider the solution $S(X) = \epsilon$): when X is to be replaced with $a^\ell X$ and $S(X)$ is a solution of the old equation then the new equation has a corresponding solution $S'(X)$ *unless* $S(X) = a^\ell$. So before the replacement, it is tested whether $S(X) = a^\ell$ is a solution and if so, it is reported. The test itself is simple: both sides of the equation are read and their values under substitution $S(X) = a^\ell$ are created on the fly and compared symbol by symbol, until a mismatch is found or both strings end.

It is easy to implement the recompression so that one phase takes linear time. Then the cost can be distributed to explicit words between the variables, each of them is charged proportionally to its length. Consider such a string w, if it is long enough, its length decreases by a constant factor in one phase, see Lemma 10. Thus, the cost of compressing this fragment and testing a solution can be charged to the lost length. However, this is not true when w is short and the $\#_X \log n$ summand in the running time comes from bounding the running time for such 'short' strings.

In Sect. 4 it is shown that using a couple of heuristics as well as more involved analysis the running time can be lowered to $\mathcal{O}(n)$. The mentioned heuristics are as follows:

- The problematic 'short' words between the variables need to be substrings of the 'long' words, this allows smaller storage size and consequently faster compression.
- When we compare $Xw_1Xw_2 \ldots w_mX$ from one side of the equation with its copy (i.e. another occurrence of $Xw_1Xw_2 \ldots w_mX$) on the other side, we make such a comparison in $\mathcal{O}(1)$ time (using suffix arrays).
- $(S(X)u)^m$ and $(S(X)u')^{m'}$ (perhaps offsetted) are compared in $\mathcal{O}(|u| + |u'|)$ time instead of naive $\mathcal{O}(m \cdot |u| + m' \cdot |u'|)$, using simple facts from combinatorics on words (i.e. periodicity).

Furthermore a more insightful analysis shows that problematic 'short' words in the equation can be used to invalidate several candidate solutions fast, even before a mismatch in the equation is found during the testing. This allows a tighter estimation of the time spent on testing the solutions.

A Note on the Computational Model

In order to perform the recompression efficiently, some algorithm for grouping pairs (and blocks) is needed. When we can identify the symbols in Γ with consecutive numbers, the grouping can be done using RadixSort in linear time. Thus, all (efficient) applications of recompression technique make such an assumption. On the other hand, the second of the mentioned heuristics requires checking string equality in constant

time, to this end a suffix array [11] plus a structure for answering *longest common prefix query* (lcp) [12] are employed and we use range minimum queries [1] on them. The last structure needs the flexibility of the RAM model to run in $\mathcal{O}(1)$ time per query.

2 Preliminaries

2.1 One-Variable Equations

A *word equation* with one-variable over the alphabet Γ and variable X is '$\mathcal{A} = \mathcal{B}$', where $\mathcal{A}, \mathcal{B} \in (\Gamma \cup \{X\})^*$. During the run of algorithm OneVarWordEq we introduce new letters into Γ, but no new variable is introduced. In this paper we shall consider only equations with one-variable.

Without loss of generality in a word equation $\mathcal{A} = \mathcal{B}$ one of $\mathcal{A}$ and $\mathcal{B}$ begins with a variable and the other with a letter:

- if they both begins with the same symbol (be it letter or nonterminal), we can remove this symbol from them, without affecting the set of solutions;
- if they begin with different letters, this equation clearly has no solution.

The same applies to the last symbols of $\mathcal{A}$ and $\mathcal{B}$. Thus, in the following we assume that the equation is of the form

$$A_0 X A_1 \dots A_{n_\mathcal{A}-1} X A_{n_\mathcal{A}} = X B_1 \dots B_{n_\mathcal{B}-1} X B_{n_\mathcal{B}}, \tag{1}$$

where $A_i, B_i \in \Gamma^*$ (we call them *words* or *explicit words*) and $n_\mathcal{A}$ ($n_\mathcal{B}$) denote the number of X occurrences in $\mathcal{A}$ ($\mathcal{B}$, respectively). Note that exactly one of $A_{n_\mathcal{A}}$, $B_{n_\mathcal{B}}$ is empty and A_0 is non-empty. If this condition is violated for any reason, we greedily repair it by cutting identical letters (or variables) from both sides of the equation. We say that A_0 is the *first* word of the equation and the non-empty of $A_{n_\mathcal{A}}$ and $B_{n_\mathcal{B}}$ is the *last word*. We additionally assume that none of words A_i, B_j is empty. We later (after Lemma 4) justify why this is indeed without loss of generality.

A *substitution* S assigns a string to X, we expand it to $(X \cup \Gamma)^*$ with an obvious meaning. A *solution* is a substitution such that $S(\mathcal{A}) = S(\mathcal{B})$. For a given equation $\mathcal{A} = \mathcal{B}$ we are looking for a description of all its solutions. We treat the empty solution $S(X) = \epsilon$ in a special way and always assume that $S(X) \neq \epsilon$.

Note that if $S(X) \neq \epsilon$, then using (1) we can always determine the first (a) and last (b) letter of $S(X)$ in $\mathcal{O}(1)$ time. In fact, we can determine the length of the a-prefix and b-suffix of $S(X)$.

Lemma 1 *For every solution S of a word equation the first letter of $S(X)$ is the first letter of A_0 and the last the last letter of $A_{n_\mathcal{A}}$ or $B_{n_\mathcal{B}}$ (whichever is non-empty).*

If $A_0 \in a^+$ then $S(X) \in a^+$ for each solution S of $\mathcal{A} = \mathcal{B}$.

If the first letter of A_0 is a and $A_0 \notin a^+$ then there is at most one solution $S(X) \in a^+$, existence of such a solution can be tested (and its length returned) in $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ time. Furthermore, for $S(X) \notin a^+$ the lengths of the a-prefixes of $S(X)$ and A_0 are the same.

Two comments are in place:

- Symmetric version of Lemma 1 holds for the suffix of $S(X)$.
- It is later shown that finding all solutions from a^+ can be done in linear time, see Lemma 11.

Proof Concerning the first claim, observe that the first letter of $S(\mathcal{A})$ is the first letter of A_0, while the first letter of $S(\mathcal{B})$ is the first letter of $S(X)$, hence those letters are equal. The same applies to the last letter of $S(X)$ and the last letter of $A_{n_\mathcal{A}}$ or $B_{n_\mathcal{B}}$, whichever of them is non-empty.

Consider the case when $A_0 \in a^+$ and suppose that $S(X) \notin a^*$, let $\ell \geq 0$ be the length of the a-prefix of $S(X)$. The length of the a-prefix of $S(\mathcal{A})$ is then $|A_0| + \ell > \ell$, which is the length of the a-prefix of $S(\mathcal{B})$, contradiction. Hence $S(X) \in a^+$.

Consider now the case when A_0 begins with a but $A_0 \notin a^+$, let its a-prefix have a length ℓ_A. Consider $S(X) \in a^+$, say $S(X) = a^\ell$. Let the first letter other than a in $\mathcal{B}$ be the $\ell_B + 1$ letter in $\mathcal{B}$ and let it be in explicit word B_i. If there is no such B_i then there is no solution $S(X) \in a^+$, as then $S(\mathcal{B})$ consists only of as, which is not true for $S(\mathcal{A})$. The length of the a-prefix of $S(\mathcal{A})$ is ℓ_A, while the length of the a-prefix of $S(\mathcal{B})$ is $\ell_B + i \cdot \ell$. Those two need to be equal, so $\ell_A = \ell_B + i \cdot \ell$ and consequently $\ell = \frac{\ell_A - \ell_B}{i}$, so this is the only candidate for the solution.

It is easy to verify whether $S(X) = a^\ell$ is a solution for a single ℓ in linear time. It is enough to compare $S(\mathcal{A})$ and $S(\mathcal{B})$ letter by letter, note that they can be created on the fly while reading $\mathcal{A}$ and $\mathcal{B}$. Each such comparison consumes one symbol from $\mathcal{A}$ and $\mathcal{B}$ (note that if we compare a suffix of $S(X)$, i.e. some $a^{\ell'}$ for $\ell' < \ell$, with $S(X) = a^\ell$ we simply remove $a^{\ell'}$ from both those strings). So the running time is linear.

Lastly, consider $S(X) \notin a^*$. Then the a-prefix of $S(\mathcal{A})$ has length ℓ_A and since $S(X) \notin a^+$, the a-prefix of $S(\mathcal{B})$ is the same as the a-prefix of $S(X)$, which consequently has length ℓ_A. □

By TestSimpleSolution(a) we denote a procedure, described in Lemma 1, that for $A_0 \notin a^*$ establishes the unique possible solution $S(X) = a^\ell$, tests it and returns ℓ if this indeed is a solution.

2.2 Representation of Solutions

Consider any solution S of $\mathcal{A} = \mathcal{B}$. We claim that $S(X)$ is uniquely determined by its length and so when describing solution of $\mathcal{A} = \mathcal{B}$ it is enough to give their lengths.

Lemma 2 *Each solution S of equation of the form (1) is of the form $S(X) = (A_0)^k A$, where A is a prefix of A_0 and $k \geq 0$. In particular, it is uniquely defined by its length.*

Proof If $|S(X)| \leq |A_0|$ then $S(X)$ is a prefix of A_0. When $|S(X)| > |A_0|$ then $S(\mathcal{A})$ begins with $A_0 S(X)$ while $S(\mathcal{B})$ begins with $S(X)$ and thus $S(X)$ has a period A_0. Consequently, it is of the form $A_0^k A$, where A is a prefix of A_0. □

Weight

Each letter in the current instance of our algorithm OneVarWordEq represents some string (in a compressed form) of the input equation, we store its *weight* which is the length of such a string. Furthermore, when we replace X with $a^\ell X$ (or Xa^ℓ) we keep track of the sum of weights of all letters removed so far from X. In this way, for each solution of the current equation we know what is the length of the corresponding solution of the original equation (it is the sum of weights of letters removed so far from X and the weight of the current solution). Therefore, in the following, we will not explain how we recreate the solutions of the original equation from the solution of the current one. Concerning the running time needed to calculate the length of the original solution: our algorithm OneVarWordEq reports only solutions of the form a^ℓ, so we just need to multiply ℓ with the weight of a and add the weights of the removed suffix and prefix.

2.3 Recompression

We recall here the technique of recompression [5,7,8], restating all important facts about it. Note that in case of one-variable many notions simplify.

2.3.1 Preserving Solutions

All subprocedures of the presented algorithm should preserve solutions, i.e. there should be a one-to-one correspondence between solution before and after the application of the subprocedure. However, when we replace X with $a^\ell X$ (or Xb^r), some solutions may be lost in the process and so they should be reported. We formalise these notions.

Definition 1 (*Preserving solutions*) A subprocedure *preserves solutions* when given an equation $\mathcal{A} = \mathcal{B}$ it returns $\mathcal{A}' = \mathcal{B}'$ such that for some strings u and v (calculated by the subprocedure)

- some solutions of $\mathcal{A} = \mathcal{B}$ are reported by the subprocedure;
- for each unreported solution S of $\mathcal{A} = \mathcal{B}$ there is a solution S' of $\mathcal{A}' = \mathcal{B}'$, where $S(X) = uS'(X)v$ and $S(\mathcal{A}) = uS'(\mathcal{A}')v$;
- for each solution S' of $\mathcal{A}' = \mathcal{B}'$ the $S(X) = uS'(X)v$ is an unreported solution of $\mathcal{A} = \mathcal{B}$ and additionally $S(\mathcal{A}) = uS'(\mathcal{A}')v$.

The intuitive meaning of these conditions is that during transformation of the equation, either we report a solution or the new equation has a corresponding solution (and no new 'extra' solutions).

By $\mathrm{h}_{c\to ab}(w)$ we denote the string obtained from w by replacing each c by ab, which corresponds to the inverse of pair compression. We say that a subprocedure *implements pair compression* for ab, if it satisfies the conditions from Definition 1, but with $S(X) = u\,\mathrm{h}_{c\to ab}(S'(X))v$ and $S(\mathcal{A}) = u\,\mathrm{h}_{c\to ab}(S'(\mathcal{A}'))v$ replacing $S(X) = uS'(X)v$ and $S(\mathcal{A}) = uS'(\mathcal{A}')v$.

Similarly, by $\mathrm{h}_{\{a_\ell \to a^\ell\}_{\ell>1}}(w)$ we denote the string w with letters a_ℓ replaced with blocks a^ℓ, for each $\ell > 1$; note that this requires that we know, which letters 'are' a_ℓ and what is the value of ℓ, but this is always clear from the context. A notion of *implementing blocks compression* for a letter a is defined similarly as the notion of implementing pair compression. The intuitive meaning of both those notions is the same as in case of preserving solutions: we not loose, nor gain any solutions.

Note that a composition of operations preserving solutions also preserves solution and a composition of an operation preserving the solution (first) and a one that implements pair compression (second) also implements the pair compression; the same applies to a composition of operations preserving a solution and implementing block compression.

Given an equation $\mathcal{A} = \mathcal{B}$, its solution S and a pair $ab \in \Gamma^2$ occurring $S(\mathcal{A})$ (or $S(\mathcal{B})$) we say that this occurrence is *explicit*, if it comes from substring ab of $\mathcal{A}$ (or $\mathcal{B}$, respectively); *implicit*, if it comes (wholly) from $S(X)$; *crossing* otherwise. A pair is *crossing* if it has a crossing occurrence and *non-crossing* otherwise. Similar notion applies to maximal blocks of as, in which case we say that *a has a crossing block* or it *has no crossing blocks*. Alternatively, a pair ab is crossing if b is the first letter of $S(X)$ and aX occurs in the equation or a is the last letter of $S(X)$ and Xb occurs in the equation or a is the last and b the first letter of $S(X)$ and XX occurs in the equation. Similar reformulation applies to crossing blocks, with a taking the place of both a and b.

Unless explicitly stated, we consider crossing/non-crossing pairs ab for $a \neq b$. Note that as the first (last) letter of $S(X)$ is the same for each S, see Lemma 1, the definition of the crossing pair *does not depend on the solution*; the same applies to crossing blocks.

When a pair ab is non-crossing, its compression is easy, as it is enough to replace each explicit ab with a fresh letter c

Algorithm 1 PairCompNCr(a, b) Pair compression for a non-crossing pair

1: let $c \in \Gamma$ be an unused letter
2: replace each explicit ab in $\mathcal{A}$ and $\mathcal{B}$ by c

Similarly when none block of a has a crossing occurrence, the a's blocks compression consists simply of replacing explicit a blocks.

Algorithm 2 BlockCompNCr(a) Block compression for a letter a with no crossing block

1: **for** $\ell > 1$ **do**
2: **for** each explicit a's ℓ-block occurring in $\mathcal{A}$ or $\mathcal{B}$ **do**
3: let $a_\ell \in \Gamma$ be an unused letter
4: replace every explicit a's ℓ-block occurring in $\mathcal{A}$ or $\mathcal{B}$ by a_ℓ

Lemma 3 *Let ab be a non-crossing pair then* PairCompNCr(a, b) *implements the pair compression for ab. Let a has no crossing blocks, then* BlockCompNCr(a) *implements the block compression for a.*

Proof Consider first the case of PairCompNCr. Suppose that $\mathcal{A} = \mathcal{B}$ has a solution S. Define S': $S'(X)$ is equal to $S(X)$ with each ab replaced with c (where c is a new letter). Consider $S(\mathcal{A})$ and $S'(\mathcal{A}')$. Then $S'(\mathcal{A}')$ is obtained from $S(\mathcal{A})$ by replacing each ab with c (as $a \neq b$ this is well-defined): the explicit occurrences of ab are replaced by PairCompNCr(a, b), the implicit ones are replaced by the definition of S' and by the assumption there are no crossing occurrences. The same applies to $S(\mathcal{B})$ and $S'(\mathcal{B}')$, hence S' is a solution of $\mathcal{A}' = \mathcal{B}'$.

Since c is a fresh letter, the $S(\mathcal{A})$ is obtained from $S'(\mathcal{A}')$ by replacing each c with ab, the same applies to $S(X)$ and $S'(X)$ as well as $S(\mathcal{B})$ and $S'(\mathcal{B}')$. Hence $S(\mathcal{A}) = \mathrm{h}_{c\to ab}(S'(\mathcal{A})) = \mathrm{h}_{c\to ab}(S'(\mathcal{B})) = S(\mathcal{B})$ and $S(X) = \mathrm{h}_{c\to ab}(S'(X))$, as required by the definition of implementing the pair compression.

Lastly, for a solution S' of $\mathcal{A}' = \mathcal{B}'$ take the corresponding S defined as $S(X) = \mathrm{h}_{c\to ab}(S'(X))$ (i.e. replacing each c with ab in $S'(X)$). It can be easily shown that $S(\mathcal{A}) = \mathrm{h}_{c\to ab}(S'(\mathcal{A}'))$ and $S(\mathcal{B}) = \mathrm{h}_{c\to ab}(S'(\mathcal{B}'))$, thus S is a solution of $\mathcal{A} = \mathcal{B}$.

The proof for the block compression follows in the same way. □

The main idea of the recompression method is the way it deals with the crossing pairs: imagine ab is a crossing pair, this is because $S(X) = bw$ and aX occurs in $\mathcal{A} = \mathcal{B}$ or $S(X) = wa$ and Xb occurs in it (the remaining case, in which $S(X) = bwa$ and XX occurs in the equation is treated in the same way). The cases are symmetric, so we deal only with the first one. To 'uncross' ab in this case it is enough to 'left-pop' b from X: replace each X in the equation with bX and implicitly change the solution to $S(X) = w$. Note that before replacing X with bX we need to check, whether $S(X) = b$ is a solution, as this solution cannot be represented in the new equation; similar remark applies to replacing X with Xa.

Algorithm 3 Pop(a, b)

1: **if** b is the first letter of $S(X)$ **then**
2: **if** TestSimpleSolution(b) returns 1 **then** ▷ $S(X) = b$ is a solution
3: report solution $S(X) = b$
4: replace each X in $\mathcal{A} = \mathcal{B}$ by bX
▷ Implicitly change $S(X) = bw$ to $S(X) = w$
5: **if** a is the last letter of $S(X)$ **then**
6: **if** TestSimpleSolution(a) returns 1 **then** ▷ $S(X) = a$ is a solution
7: report solution $S(X) = a$
8: replace each X in $\mathcal{A} = \mathcal{B}$ by Xa
▷ Implicitly change $S(X) = w'a$ to $S(X) = w'$

Lemma 4 Pop(a, b) *preserves solutions and after its application the pair ab is non-crossing.*

Note that Lemma 4 justifies our earlier claim that without loss of generality we can assume that none of A_i, B_j is empty: at the beginning of the algorithm we can run $\mathsf{Pop}(a, b)$ once for a being the first letter of $S(X)$. This ensures the claim and increases the size of the instance at most twice.

Proof It is easy to verify that a pair ab is crossing if and only if one of the following situations occurs:

CP1 aX occurs in the equation and the first letter of $S(X)$ is b;
CP2 Xb occurs in the equation and the last letter of $S(X)$ is a;
CP3 XX occurs in the equation, the first letter of $S(X)$ is b and the last a.

Let $\mathcal{A}' = \mathcal{B}'$ be the obtained equation, we show that ab in $\mathcal{A}' = \mathcal{B}'$ is noncrossing. Consider whether X was replaced by bX is line 4. If not, then the first letter of $S(X)$ and $S'(X)$ is not b, so neither (CP1) nor (CP3) hold. Suppose that X was replaced with bX. Then to the left of each X there is a letter which is not a, so none of situations (CP1), (CP3) occurs.

A similar analysis applied to the last letter of $S(X)$ yields that (CP2) cannot happen and so ab cannot be a crossing pair.

Pop can be naturally divided into two parts, which correspond to the replacement of X by bX and the replacement of X by Xa. We show for the first one that it preserves solutions, the proof for the second one is identical.

If $S(X)$ does not begin with b (recall that all solutions have the same first letter, see Lemma 1) then nothing changes and the set of solutions is preserved. Otherwise $S(X) = bw$ and there are two subcases:

if $w = \epsilon$ then it is reported in line 3;
if $w \neq \epsilon$ then it is not reported and $S'(X) = w$ is a solution of the obtained equation.

Moreover, if b is reported then indeed it is a solution. On the other hand, whenever $S'(X) = w$ is a solution after popping b then $S(X) = bw$ is a solution of $\mathcal{A} = \mathcal{B}$.

A symmetric analysis is done for the operation of right-popping a, which ends the proof. □

Now the presented procedures can be merged into one procedure that turns crossing pairs into noncrossing ones and then compresses them, effectively compressing crossing pairs.

Algorithm 4 $\mathsf{PairComp}(a, b)$ Turning crossing pair ab into non-crossing ones and compressing it

1: run $\mathsf{Pop}(a, b)$
2: run $\mathsf{PairCompNCr}(b, a)$

Lemma 5 $\mathsf{PairComp}(a, b)$ *implements the pair compression of the pair* ab.

The proof follows by combining Lemmas 3 and 4.

There is one issue: the number of non-crossing pairs can be large, however, a simple preprocessing, which basically applies Pop, is enough to reduce the number of crossing pairs to 2.

Algorithm 5 PreProc Ensures that there are at most 2 crossing pairs

1: let a, b be the first and last letter of $S(X)$
2: run Pop(a, b)

Lemma 6 PreProc *preserves solution and after its application there are at most two crossing pairs.*

Proof It is enough to show that there are at most 2 crossing pairs, as the rest follows from Lemma 4. Let a and b be the first and last letters of $S(X)$, and a', b' such letters after the application of PreProc. Then each X is preceded with a and succeeded with b in $\mathcal{A}' = \mathcal{B}'$. So the only crossing pairs are aa' and $b'b$ (note that this might be the same pair or part of a letter-block, i.e. $a = a'$ or $b = b'$). □

The problems with crossing blocks can be solved in a similar fashion: a has a crossing block if and only if aa is a crossing pair. So we 'left-pop' a from X until the first letter of $S(X)$ is different than a, we do the same with the ending letter b. This can be alternatively seen as removing the whole a-prefix (b-suffix, respectively) from X: suppose that $S(X) = a^\ell w b^r$, where w does not begin with a nor end with b. Then we replace each X by $a^\ell X b^r$ implicitly changing the solution to $S(X) = w$, see Algorithm 6.

Algorithm 6 CutPrefSuff Cutting prefixes and suffixes; assumes that A_0 is not a block of letters

Require: A_0 is not a block of letters, the non-empty of $A_{n_\mathcal{A}}$, $B_{n_\mathcal{B}}$ is not a block of letters

1: let a be the first letter of $S(X)$
2: report solution found by TestSimpleSolution(a)
▷ Excludes $S(X) \in a^+$ from further considerations.
3: let $\ell > 0$ be the length of the a-prefix of A_0
▷ By Lemma 1 $S(X)$ has the same a-prefix
4: replace each X in $\mathcal{A} = \mathcal{B}$ by $a^\ell X$ ▷ a^ℓ is stored in a compressed form,
▷ implicitly change $S(X) = a^\ell w$ to $S(X) = w$
5: let b be the last letter of $S(X)$
6: report solution found by TestSimpleSolution(b)
▷ Exclude $S(X) \in b^+$ from further considerations.
7: let $r > 0$ be the length of the b-suffix of the non-empty of $A_{n_\mathcal{A}}$, $B_{n_\mathcal{B}}$
▷ By Lemma 1 $S(X)$ has the same b-suffix
8: replace each X in $\mathcal{A} = \mathcal{B}$ by Xb^r ▷ b^r is stored in a compressed form,
▷ implicitly change $S(X) = w b^r$ to $S(X) = w$

Note that in order to claim that the lengths of a-prefix of $S(X)$ and A_0 are the same, see Lemma 1, we need to assume that $S(X)$ is a not block of letters. This is fine though, as this condition holds when we apply Algorithm 6.

Lemma 7 *Let a be the first letter of the first word and b the last of the last word. If the first word is not a block of as and the last not a block of bs then* CutPrefSuff *preserves solutions and after its application there are no crossing blocks of letters.*

Proof Consider first only the changes done by the modification of the prefix. Suppose that $S(X) = a^\ell w$, where w does not begin with a. If $w = \epsilon$ then, as $A_0 \notin a^+$ from the assumption, by Lemma 1 there is only one such solution and it is reported in line 2. Otherwise, by Lemma 1, each solution S of the equation is of the form $S(X) = a^\ell w$, where a^ℓ is the a-prefix of A_0 and $w \neq \epsilon$ nor w does begin with a. Then the $S'(X) = w$ is the solution of the new equation. Similarly, for any solution $S'(X) = w$ the $S(X) = a^\ell w$ is the solution of the original equation.

The same analysis can be applied to the modifications of the suffix: observe that if at the beginning the last word was not a block of bs it did not become one during the cutting of the a-prefix.

Lastly, suppose that some letter c has a crossing block, without loss of generality assume that c is the first letter of $S(X)$ and cX occurs in the equation. But this is not possible: X was replaced by $a^\ell X$ and so the only letter to the left of X is a and $S(X)$ does not begin with a, contradiction. □

The CutPrefSuff allows defining a procedure BlockComp that compresses maximal blocks of all letters, regardless of whether they have crossing blocks or not.

Algorithm 7 BlockComp Compressing blocks of a

```
1: Letters ← letters occurring in the equation
2: run CutPrefSuff                          ▷ Removes crossing blocks of a
3: for each letter a ∈ Letters do
4:     BlockCompNCr(a)
```

Lemma 8 *Let a be the first letter of the first word and b the last of the last word. If the first word is not a block of as and the last not a block of bs then* BlockComp *implements the block compression for letters present in $\mathcal{A} = \mathcal{B}$ before its application.*

The proof follows by combining Lemmas 3 and 7.

3 Main Algorithm

The following algorithm OneVarWordEq is basically a specialisation of the general algorithm for testing the satisfiability of word equations [7] and is built up from procedures presented in the previous section.

Algorithm 8 OneVarWordEq Reports solutions of a given one-variable word equation

```
1: while the first block and the last block are not blocks of a letter do
2:     Pairs ← pairs occurring in S(A) = S(B)
3:     BlockComp                          ▷ Compress blocks, in O(|A| + |B|) time.
4:     PreProc                            ▷ There are only two crossing pairs, see Lemma 6
5:     Crossing ← list of crossing pairs from Pairs      ▷ There are two such pairs
6:     Non-Crossing ← list of non-crossing pairs from Pairs
7:     for each ab ∈ Non-Crossing do
                          ▷ Compress non-crossing pairs, in time O(|A| + |B)|
8:         PairCompNCr(a, b)
9:     for ab ∈ Crossing do   ▷ Compress the 2 crossing pairs, in time O(|A| + |B)|
10:        PairComp(a, b)
11: TestSolution                          ▷ Test solutions from a*, see Lemma 11
```

We call one iteration of the main loop of OneVarWordEq a *phase*.

Theorem 1 OneVarWordEq *runs in time* $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}| + (n_\mathcal{A} + n_\mathcal{B}) \log(|\mathcal{A}| + |\mathcal{B}|))$ *and correctly reports all solutions of a word equation* $\mathcal{A} = \mathcal{B}$.

Before showing the running time, let us first comment on how the equation is stored. Each of the sides ($\mathcal{A}$ and $\mathcal{B}$) is represented as two lists of pointers to strings, i.e. to $A_0, A_1, \ldots, A_{n_\mathcal{A}}$ and to $B_0, B_1, \ldots, B_{n_\mathcal{B}}$. Each of those words is stored as a doubly-linked list. When we want to refer to a concrete word in a phase, we use names A_i and B_j, when we want to stress its evolution in phases, we use names $(\mathcal{A}, i)$-word and $(\mathcal{B}, j)$-word.

Shortening of the Solutions

The most important property of OneVarWordEq is that the explicit strings between the variables shorten (assuming that they have a large enough length). To show this we use the following technical lemma, which is also used several times later on:

Lemma 9 *Consider two consecutive letters* a, b *at the beginning of the phase in* $S(\mathcal{A})$ *for any solution* S. *At least one of those letters is compressed in this phase.*

Proof Consider whether $a = b$ or not:

- $a = b$: In this case they are compressed using BlockComp.
- $a \neq b$: In this case ab is a pair occurring in the equation at the beginning of the phase and so it was listed in *Pairs* in line 2 and as such we try to compress it, either in line 8 or in line 10. This occurrence cannot be compressed only when one of the letters a, b was already compressed, in some other pair or by BlockComp. In either case we are done. □

We say that a word A_i (B_i) is *short* if it consists of at most 100 letters and *long* otherwise. To avoid usage of strange constants and its multiplicities, we shall use $N = 100$ to denote this value and we shall usually say that $N = \mathcal{O}(1)$.

Lemma 10 *Consider the length of the* $(\mathcal{A}, i)$*-word (or* $(\mathcal{B}, j)$*-word). If it is long then its length is reduced by* $1/4$ *in this phase. If it is short then after the phase it still is. The length of each unreported solution is reduced by at least* $1/4$ *in a phase.*

Additionally, if the first (last) word is short and has at least 2 *letters then its length is shortened by at least* 1 *in a phase.*

Proof We shall first deal with the words and then comment how this argument extends to the solutions. Consider two consecutive letters a, b in any word at the beginning of a phase. By Lemma 9 at least one of those letters is compressed in this phase. Hence each uncompressed letter in a word (except perhaps the last letter) can be associated with the two letters to the right that are compressed. This means that in a word of length k during the phase at least $\frac{2(k-1)}{3}$ letters are compressed i.e. its length is reduced by at least $\frac{k-1}{3}$ letters.

On the other hand, letters are introduced into words by popping them from variables. Let *symbol* denote a single letter or block a^ℓ that is popped into a word. We investigate, how many symbols are introduced in this way in one phase. At most one symbol is popped to the left and one to the right by BlockComp in line 3, the same holds for PreProc in line 4. Moreover, one symbol is popped to the left and one to the right in line 10; since this line is executed twice, this yields 8 symbols in total. Note that the symbols popped by BlockComp are replaced by single letters, so the claim in fact holds for letters as well.

So, consider any word $A_i \in \Gamma^*$ (the proof for B_j is the same), at the beginning of the phase and let A'_i be the corresponding word at the end of the phase. There were at most 8 symbols introduced into A'_i (some of them might be compressed later). On the other hand, by Lemma 9, at least $\frac{|A_i|-1}{3}$ letters were removed A_i due to compression. Hence

$$|A'_i| \le |A_i| - \frac{|A_i|-1}{3} + 8 \le \frac{2|A_i|}{3} + 8\frac{1}{3}.$$

It is easy to check that when A_i is short, i.e. $|A_i| \le N = 100$, then A'_i is short as well and when A_i is long, i.e. $|A_i| > N$ then $|A'_i| \le \frac{3}{4}|A_i|$.

It is left to show that the first word shortens by at least one letter in each phase. Consider that if a letter a is left-popped from X then we created B_0 and in order to preserve (1) the first letters of B_0 and A_0 are removed. Thus, A_0 gained one letter on the right and lost one on the left, so its length stayed the same. Furthermore the right-popping does not affect the first word at all (as X is not to its left); the same analysis applies to cutting the prefixes and suffixes. Hence the length of the first word is never increased by popping letters. Moreover, if at least one compression (be it block compression or pair compression) is performed inside the first word, its length drops. So consider the first word at the end of the phase let it be A_0. Note that there is no letter representing a compressed pair or block in A_0: consider for the sake of contradiction the first such letter that occurred in the first word. It could not occur through a compression inside the first word (as we assumed that it did not happen), cutting prefixes does not introduce compressed letters, nor does popping letters. So in A_0 there are no compressed letters. But if $|A_0| > 1$ then this contradicts Lemma 9.

Now, consider a solution $S(X)$. We know that $S(X)$ is either a prefix of A_0 or of the form $A_0^\ell A$, where A is a prefix of A_0, see Lemma 2. In the former case, $S(X)$ is compressed as a substring of A_0. In the latter observe that argument follows in the same way, as long as we try to compress every pair of letters in $S(X)$. So consider such a pair ab. If it is inside A_0 then we are done. Otherwise, a is the last letter of A_0 and b the first. Then this pair occurs also on the crossing between A_0 and X in $\mathcal{A}$, i.e. ab is one of the crossing pairs. In particular, we try to compress it. So, the claim of the lemma holds for $S(X)$ as well. □

The correctness of the algorithm follows from Lemmata 8 (for BlockComp), Lemma 6 (for PreProc), Lemma 3 (for PairCompNCr), Lemma 5 (for PairComp) and from the lemma below, which deals with TestSolution.

Lemma 11 *For $a \in \Gamma$ we can report all solutions in which $S(X) = a^\ell$ for some natural ℓ in $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ time. There is either exactly one ℓ for which $S(X) = a^\ell$ is a solution or $S(X) = a^\ell$ is a solution for each ℓ or there is no solution of this form.*

Note that we do not assume that the first or last word is a block of as.

Proof The algorithm and proof is similar as in Lemma 1. Consider a substitution $S(X) = a^\ell$. We calculate the length of the a-prefix of $S(\mathcal{A})$ and $S(\mathcal{B})$. Consider first letter other than a in $\mathcal{A}$, let it be in the A_{k_A} and suppose that there were ℓ_A letters a before it (if there is non such letter, imagine we attach an 'ending marker' to both $\mathcal{A}$ and $\mathcal{B}$, which then becomes such a letter). Then the length of the a-prefix of $S(\mathcal{A})$ is $k_A \cdot \ell + \ell_A$. Let additionally $\mathcal{A}'$ be obtained from $\mathcal{A}$ by removing those letters a and variables in between them. Similarly, define k_B, ℓ_B and $\mathcal{B}'$. Then the length of the a-prefix of $S(\mathcal{B})$ is $k_B \cdot \ell + \ell_B$.

The substitution $S(X) = a^\ell$ is a solution if and only if $k_A \cdot \ell + \ell_A = k_B \cdot \ell + \ell_B$ and $S(\mathcal{A}') = S(\mathcal{B}')$. Consider the number of natural solutions of the equation

$$k_A \cdot x + \ell_A = k_B \cdot x + \ell_B :$$

- *no natural solution:* Clearly there is no solution of the word equation $\mathcal{A} = \mathcal{B}$.
- *one solution $x = \ell$*: Then $S(X) = a^\ell$ is the only possible solution from a^+ of $\mathcal{A} = \mathcal{B}$. To verify whether S satisfies $\mathcal{A}' = \mathcal{B}'$ we apply the same strategy as in TestSimpleSolution(a): we evaluate both sides of $\mathcal{A}' = \mathcal{B}'$ under the substitution $S(X) = a^\ell$ on the fly. The same argument as in Lemma 1 shows that the running time is linear in $|\mathcal{A}'| + |\mathcal{B}'|$.
- *satisfied by all natural numbers:* Then the a-prefixes of $\mathcal{A}$ and $\mathcal{B}$ are of the same length for each $S(X) \in a^*$. We thus repeat the procedure for $\mathcal{A}' = \mathcal{B}'$, shortening them so that they obey the form (1), if needed. Clearly, solutions in a^* of $\mathcal{A}' = \mathcal{B}'$ are exactly the solutions of $\mathcal{A} = \mathcal{B}$ in a^*.

The stopping condition for the recurrence above is obvious: if $\mathcal{A}'$ and $\mathcal{B}'$ are both empty then we are done (each $S(X) = a^\ell$ is a solution of this equation), if exactly one of them is empty and the other is not then there is no solution at all.

Lastly, observe that the cost of the subprocedure above is proportional to the amount of read letters, which are then not read again, so the running time is $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$. □

Running Time

Concerning the running time, we first show that one phase runs in linear time, which follows by standard approach, and then that in total the running time is $\mathcal{O}(n+\#_X \log n)$. To this end we assign in a fixed phase to each $(\mathcal{A}, i)$-word and $(\mathcal{B}, j)$-word cost proportional to their lengths in this phase. For a fixed $(\mathcal{A}, i)$-word the sum of costs assigned while it was long forms a geometric sequence, so sums up to at most constant more than the initial length of $(\mathcal{A}, i)$-word; on the other hand the cost assigned when $(\mathcal{A}, i)$-word is short is $\mathcal{O}(1)$ per phase and there are $\mathcal{O}(\log n)$ phases.

Lemma 12 *One phase of* OneVarWordEq *can be performed in* $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ *time.*

Proof For grouping of pairs and blocks we use RadixSort, to this end it is needed that the alphabet of (used) letters can be identified with consecutive numbers, i.e. with an interval of at most $|\mathcal{A}| + |\mathcal{B}|$ integers. In the first phase of OneVarWordEq this follows from the assumption on the input.[1] At the end of this proof we describe how to bring back this property at the end of the phase.

To perform BlockComp we want for each letter a occurring in the equation to have lists of all maximal a-blocks occurring in $\mathcal{A} = \mathcal{B}$ (note that after CutPrefSuff there are no crossing blocks, see Lemma 7). This is done by reading $\mathcal{A} = \mathcal{B}$ and listing triples (a, k, p), where k is the length of a maximal block of as and p is a pointer to the beginning of this occurrence. Notice, that the maximal block of a's may consist also of prefixes/suffixes that were cut from X by CutPrefSuff. However, by Lemma 1 such a prefix is of length at most $|A_0| \leq |\mathcal{A}| + |\mathcal{B}|$ (and similar analysis applies for the suffix). Then each maximal block includes at most one such prefix and one such suffix thus the length of the a maximal block is at most $3(|\mathcal{A}|+|\mathcal{B}|)$. Hence, the triples (a, k, p) can be sorted by their first two coordinates using RadixSort in total time $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$.

After the sorting, we go through the list of maximal blocks. For a fixed letter a, we use the pointers to localise a's blocks in the rules and we replace each of its maximal block of length $\ell > 1$ by a fresh letter. Since the blocks of a are sorted, all blocks of the same length are consecutive on the list, and replacing them by the same letter is easily done.

To compress all non-crossing pairs, i.e. to perform the loop in line 8, we do a similar thing as for blocks: we read both $\mathcal{A}$ and $\mathcal{B}$, whenever we read a pair ab where $a \neq b$ and both a and b are not letters that replaced blocks during the blocks compression, we add a triple (a, b, p) to the temporary list, where p is a pointer to this position. Then we sort all these pairs according to lexicographic order on first two coordinates, we use RadixSort for that. Since in each phase we number the letters occurring in $\mathcal{A} = \mathcal{B}$ using consecutive numbers, this can be done in time $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$. The occurrences of the crossing pairs can be removed from the list: by Lemma 6 there are at most two crossing pairs and they can be easily established (by looking at $A_0 X A_1$). So we read the sorted list of pairs occurrences and we remove from it the ones that correspond

[1] In fact, this assumption can be weakened a little: it is enough to assume that $\Gamma \subseteq \{1, 2, \ldots, \mathsf{poly}(|\mathcal{A}| + |\mathcal{B}|)\}$: in such case we can use RadixSort to sort Γ in time $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ and then replace Γ with set of consecutive natural numbers.

to a crossing pair. Lastly, we go through this list and replaces pairs, as in the case of blocks. Note that when we try to replace ab it might be that this pair is no longer there as one of its letters was already replaced, in such a case we do nothing. This situation is easy to identify: before replacing the pair we check whether it is indeed ab that we expect there, as we know a and b, this is done in constant time.

We can compress each of the crossing pairs naively in $\mathcal{O}(|\mathcal{A}|+|\mathcal{B}|)$ time by simply first applying the popping and then reading the equation form the left to the right and replacing occurrences of this fixed pair.

It is left to describe, how to enumerate (with consecutive numbers) letters in Γ at the end of each phase. Firstly notice that we can easily enumerate all letters introduced in this phase and identify them (at the end of this phase) with $\{1,\ldots,m\}$, where m is the number of introduced letters (note that none of them were removed during the OneVarWordEq). Next by the assumption the letters in Γ (from the beginning of this phase) are already identified with a subset of $\{1,\ldots,|\mathcal{A}|+|\mathcal{B}|\}$, we want to renumber them, so that the subset of letters from Γ that are present at the end of the phase is identified with $\{m+1,\ldots,m+m'\}$ for an appropriate m'. To this end we read the equation, whenever we spot a letter a that was present at the beginning of the phase we add a pair (a, p) where p is a pointer to this occurrence. We sort the list in time $\mathcal{O}(|\mathcal{A}|+|\mathcal{B}|)$. From this list we can obtain a list of present letters together with list of pointers to their occurrences in the equation. Using those pointers the renumbering is easy to perform in $\mathcal{O}(|\mathcal{A}|+|\mathcal{B}|)$ time.

So the total running time is $\mathcal{O}(|\mathcal{A}|+|\mathcal{B}|)$. □

The amortisation, especially in the next section, is much easier to be shown when we know that both the first and last words are long. This assumption is not restrictive, as as soon as one of them becomes short, the remaining running time of OneVarWordEq is linear.

Lemma 13 *As soon as first or last word becomes short, the rest of the running time of* OneVarWordEq *is* $\mathcal{O}(n)$.

Proof One phase takes $\mathcal{O}(|\mathcal{A}|+|\mathcal{B}|)$ time by Lemma 12 (this is at most $\mathcal{O}(n)$ by Lemma 10) and as Lemma 10 guarantees that both the first word and the last word are shortened by at least one letter in a phase, there will be at most $N=\mathcal{O}(1)$ many phases. Lastly, Lemma 11 shows that TestSolution also runs in $\mathcal{O}(n)$. □

So it remains to estimate the running time until one of the last or first word becomes short.

Lemma 14 *The running time of* OneVarWordEq *till one of first or last word becomes short is* $\mathcal{O}(n+(n_{\mathcal{A}}+n_{\mathcal{B}})\log n)$.

Proof By Lemma 12 the time of one iteration of OneVarWordEq is $\mathcal{O}(|\mathcal{A}|+|\mathcal{B}|)$. We distribute the cost among the $\mathcal{A}$ words and $\mathcal{B}$ words: we charge $\beta|A_i|$ to $(\mathcal{A},i)$-word and $\beta|B_j|$ to $(\mathcal{B},j)$-word, for appropriate positive β. Fix $(\mathcal{A},i)$-word, we separately estimate how much was charged to it when it was a long and short word.

– *long:* Let n_i be the initial length of $(\mathcal{A},i)$-word. Then by Lemma 10 the length in the $(k+1)$th phase it at most $(\frac{3}{4})^k n_i$ and so these costs are at most $\beta n_i+\frac{3}{4}\beta n_i+(\frac{3}{4})^2\beta n_i+\ldots\leq 4\beta n_i$.

– *short:* Since $(\mathcal{A}, i)$-wordis short, its length is at most N, so we charge at most $N\beta$ to it. Notice, that there are $\mathcal{O}(\log n)$ iterations of the loop in total, as first word is of length at most n and it shortens by $\frac{3}{4}$ in each iteration when it is long and we calculate only the cost when it is long. Hence we charge in this way $\mathcal{O}(\log n)$ times, so in total $\mathcal{O}(\log n)$.

Summing those costs over all phases and over all words and phases yields $\mathcal{O}(n + (n_{\mathcal{A}} + n_{\mathcal{B}}) \log n)$. □

4 Heuristics and Better Analysis

The intuition gained from the analysis in the previous section, especially in Lemma 14 is that the main obstacle in obtaining the linear running time is the necessity of dealing with short words, as the time spend on processing them is difficult to charge. This applies to both the compression performed within the short words, which does not guarantee any reduction in length, see Lemma 10, and to testing of the candidate solutions, which cannot be charged to the length decrease of the whole equation.

Observe that by Lemma 13 as soon as the first or last word becomes short, the remaining running time is linear. Hence, in our improvements of the running time we can restrict ourselves to the case, in which the first and last word are long.

The improvement to linear running time is done by four improvements in algorithm analysis and employed data structures, which are described in details in the following subsections:

– *several equations:* Instead of a single equation, we store a system of several equations and look for a solution of such a system. This allows removal of some words from the equations that always correspond to each other and thus decreases the overall storing space and testing time. This is described in Sects. 4.2 and 4.4.
– *small solutions:* We identify a class of particularly simple solutions, called *small*, and show that a solution is reported within $\mathcal{O}(1)$ phases from the moment when it became small. In several problematic cases of the analysis we are able to show that the solutions involved are small and so it is easier to charge the time spent on testing them. Section 4.3 is devoted to this issue.
– *storage:* The storage is changed so that all words are represented by a structure of size proportional to the size of the *long words*. In this way the storage space decreases by a constant factor in each phase and so the running time (except for testing) is linear. This is explained in Sect. 4.4.
– *testing:* The testing procedure is modified, so that the time it spends on the short words is reduced. In particular, we improve the rough estimate that one TestSimpleSolution takes time proportional to the equation to an estimation that actually counts for each word whether it was included in the test or not. Section 4.5 is devoted to this.

4.1 Suffix Arrays and lcp Arrays

We use a standard data structure for comparisons on strings: a suffix array $SA[1..m]$ for a string $w[1..m]$ stores the m non-trivial suffixes of w, i.e. $w[m], w[m-1..m], \ldots, w[1..m]$ in (increasing) lexicographical order. In other words, $SA[k] = p$ if and only if $w[p \ldots m]$ is the kth suffix according to the lexicographical order. It is known that such an array can be constructed in $\mathcal{O}(m)$ time [11] assuming that RadixSort is applicable to letters, i.e. that they are integers from $\{1, 2, \ldots, m^c\}$ for some constant c. We assume explicitly that this is the case in our problem.

Using a suffix array the equality testing for substrings of w reduces to the *longest common prefix* (lcp) query: observe that $w[i..i+k] = w[j..j+k]$ if and only if the common prefix of $w[i..m]$ and $w[j..m]$ is at least k. The first step in constructing a data structure for answering such queries is the LCP array: for each $i = 1, \ldots, m-1$ the $LCP[i]$ stores the length of the longest common prefix of $SA[i]$ and $SA[i+1]$. Given a suffix array, the LCP array can be constructed in linear time [12], however, the linear-time construction of suffix arrays can be in fact extended to return also the LCP array [11].

When the LCP array is supplied, the general longest prefix queries reduce to the range minimum queries: the longest common prefix of $SA[i]$ and $SA[j]$ (for $i < j$) is the minimum among $LCP[i], \ldots, LCP[j-1]$, and so it is enough to have a data structure that answers the queries about the minimum in the range in constant time. Such data structures in general case are known and in case of LCP arrays even simpler constructions were given [1]. The construction time is linear and query time is $\mathcal{O}(1)$ [1]. Hence, after a linear preprocessing, we can calculate the length of the longest common prefix of two substrings of a given string in $\mathcal{O}(1)$ time.

4.2 Several Equations

The improved analysis assumes that we do not store a single equation, instead, we store several equations and look for substitutions that simultaneously satisfy all of them. Hence we have a collection $\mathcal{A}_i = \mathcal{B}_i$ of equations, for $i = 1, \ldots, m$, each of them is of the form described by (1); by $\mathcal{A} = \mathcal{B}$ we denote the whole system of the equations. In particular, each of those equations specifies the first and last letter of the solution, length of the a-prefix and suffix etc., exactly in the same way as it does for a single equation. If there is a conflict, as two equations give different answers regarding the first/last letter or the length of the a-prefix or b-suffix, then there is no solution at all. Still, we do not check the consistency of all those answers, instead, we use an arbitrary equation, say $\mathcal{A}_1 = \mathcal{B}_1$, to establish the first, last letter, etc., and as soon as we find out that there is a conflict, we stop the computation and terminate immediately.

The system of equations stored by OneVarWordEq is obtained by replacing one equation $\mathcal{A}'_i\mathcal{A}''_i = \mathcal{B}'_i\mathcal{B}''_i$ (where $\mathcal{A}'_i, \mathcal{A}''_i, \mathcal{B}'_i, \mathcal{B}''_i \in (\Gamma \cup \{X\})^*$) with equivalent two equations $\mathcal{A}'_i = \mathcal{B}'_i$ and $\mathcal{A}''_i = \mathcal{B}''_i$ (note that in general the latter two equation are not equivalent to the former one, however, we perform the replacement only when they are; moreover, we need to trim them so that they satisfy the form (1).

The described way of splitting the equations implies a natural order on the equations in the system: when $\mathcal{A}'_i\mathcal{A}''_i = \mathcal{B}'_i\mathcal{B}''_i$ is split to $\mathcal{A}'_i = \mathcal{B}'_i$ and $\mathcal{A}''_i = \mathcal{B}''_i$ then $\mathcal{A}'_i = \mathcal{B}'_i$ is before $\mathcal{A}''_i = \mathcal{B}''_i$ (moreover, they are both before/after each equation before/after which $\mathcal{A}'_i\mathcal{A}''_i = \mathcal{B}'_i\mathcal{B}''_i$ was). This order is followed whenever we perform any operations on all words of the equations. We store a list of all equations, in this order.

We store each of the equations in the same way as described for a single equation in the previous phase, i.e. for an equation $\mathcal{A}_i = \mathcal{B}_i$ we store a list of pointers to words on one side and list of pointers to words on the other side. Additionally, the first word of $\mathcal{A}_i$ has a link to the last word of $\mathcal{A}_{i-1}$ and the last word of $\mathcal{A}_i$ similarly, the last word of $\mathcal{A}_i$ has a link to the first word of $\mathcal{A}_i$ and the first word of $\mathcal{A}_{i+1}$. We also say that A_i (B_j) is first or last if it is in any of the stored equations.

All operations on a single equation introduced in the previous sections (popping letters, cutting prefixes and suffixes, pair compression, blocks compression) generalise to a system of equations. The running times are addressed in detail later on. Concerning the properties, they are the same, we list those for which the generalisation or the proof are non-obvious: PreProc should ensure that there are only two crossing pairs. This is the case, as each X in every equation is replaced by the same aXb and $S(X)$ is the same for all equations, which is the main fact used in the proof of Lemma 6. Lemma 10 ensured that in each phase the length of the first and last word is decreased. Currently the first words in each equation may be different, however, the analysis in Lemma 10 applies to each of them.

4.3 Small Solutions

We say that a word w is *almost periodic* with *period size* p and *side size* s if it can be represented as $w = w_1 w_2^\ell w_3$ (where ℓ is an arbitrary number), where $|w_2| \leq p$ and $|w_1 w_3| \leq s$; we often call w_2 the *periodic part* of this factorisation. (Note that several such representation may exist, we use this notion for a particular representation that is clear from the context). A substitution S is *small*, if $S(X) = (w)^k v$, where w, v are almost periodic with period and side sizes N.

The following theorem shows the main result of this section: if a solution is small, then it is reported by OneVarWordEq within $\mathcal{O}(1)$ phases.

Theorem 2 *If $S(X)$ is a small solution then* OneVarWordEq *reports it within $\mathcal{O}(1)$ phases.*

We would like to note that the rest of the paper is independent from the proof of Theorem 2, so it might be skipped in reading.

Intuition is as follows: observe first that in each phase we make Pop and test whether $S(X) = a$, where a is a single letter, is a solution. Thus it is enough to show that a small solution is reduced to one letter within $\mathcal{O}(1)$ phases. To see this, consider first an almost periodic word, represented as $w_1 w_2^\ell w_3$. Ideally, all compressions performed in one phase of OneVarWordEq are done separately on w_1, w_3 and each w_2. In this way we obtain a string $w'_1 w'^{\ell}_2 w'_3$ and from Lemma 9 it follows that w'_i is shorter than w_i by a constant fraction. After $\mathcal{O}(\log |w_2|)$ steps we obtain a word $w''_1 w''^{\ell}_2 w''_3$ in which w''_2 is a single letter, and so in this phase w''^{ℓ}_2 is replaced with a single letter. Then,

since the length of $w_1'''w_3'''$ is at most N, after $\mathcal{O}(1)$ phases this is also reduced to a single letter. Concerning the small solution, $w^k v$ we first make such an analysis for w, when it is reduced to a single letter (after $\mathcal{O}(1)$ phases) after one additional phase $w^k = a^k$ is also reduced to one letter (by BlockComp) and so the obtained string $a_k v'$ is a concatenation of two almost periodic strings. Using the same analysis as above for each of them we obtain that it takes $\mathcal{O}(1)$ time to reduce them all to single letters. Thus we have a 2-letter string, which is reduced to a single letter within 2 phases.

In reality we need to take into the account that some compressions are made on the crossings of the considered strings, however, we can alter the factorisation (into almost periodic words and almost periodic words into periodic part and rest) of the string so that the result is almost as in the idealised case.

We say that for a substring w of $S(X)$ during one phase of OneVarWordEq the letters in w are *compressed independently*, if every compressed pair or block were either wholly within this w or wholly outside this w (in some sense this corresponds to the non-crossing compression).

The following lemma shows that given an almost periodic substring of $S(X)$ with period size p and side size s we can find an alternative representation in which the period size is the same, side size increases (a bit) but each w in w^k in this new representation is compressed independently. This shows that the intuition about shortening of almost periodic strings is almost precise—we can think that periodic part in almost periodic strings are compressed independently, but we need to pay for that by an increase in the side size.

Lemma 15 *Consider almost periodic substring of $S(X)$ with period size p and side size s represented as $w_1 w_2^\ell w_3$, where w_2 is not a block of single letter. Then there is a representation of this string as $w_1' w_2'^{\ell'} w_3'$ such that*

- $\ell - 2 \leq \ell' \leq \ell$
- $|w_2'| = |w_2|$ *(and consequently $|w_1'| + |w_3'| \leq |w_1| + |w_3| + 2|w_2|$)*
- *the form of w_2' depends solely on w_2 and does not depend on w_1, w_3 (it does depend on the equation and on the order of blocks and pairs compressed by* OneVarWordEq*)*
- *the compression in one phase of* OneVarWordEq *compresses each w' from $w'^{\ell'}$ independently.*

In particular, this other representation has period size p and side size $s + 2p$.

Proof First of all, if $\ell \leq 2$ then we take $w_2' = \epsilon, k' = 0$ and concatenate w_2^k to w_1 to obtain w_1' (and take $w_3 = w_3'$). So in the following we consider the case in which $\ell > 2$ and set $\ell' = \ell - 2$.

Let $w_2 = a^m z b^r$, where $a, b \in \Gamma, m, r \geq 1$ and $z \in \Gamma^*$ does not start with a nor it ends with b, such a representation is possible, as w_2 is not a block of letters. Note that if $a = b$ then $z \neq \epsilon$, as otherwise w_2 is a block of letters. Then $w_1 w_2^\ell w_3 = w_1 (a^m z b^r)^\ell w_3$. Since w_1 can end with a and w_2 can begin with b, we are interested in compressions within the middle $z b^r (a^m z b^r)^{\ell-2} a^m z$. We first show that indeed there is a compression of a substring that is fully within the $z b^r (a^m z b^r)^{\ell-2} a^m z$:

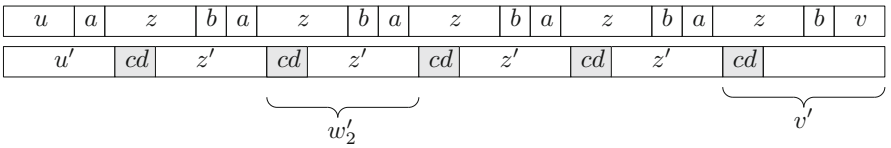

Fig. 1 The alternative factorisation. The first compressed letters are in *grey*. For simplicity $m = r = 1$. Each z' between the cds is compressed independently

- If $a = b$ then $z \neq \epsilon$ and $b^r a^m = a^{r+m}$ is a block of letters (of length at least 2) that is surrounded by z (which does not begin, nor end with a) from both ends, so it is compressed.
- If $a \neq b$ and $m > 1$ or $r > 1$ then we compress the block a^m or b^r.
- If $a \neq b, m = r = 1$ and $z = \epsilon$ then this substring is $b(ab)^{\ell-2}a$. As $\ell > 2$ the pair ab is listed by OneVarWordEq and we try to compress it. If we fail then it means that one of the letters was already compressed with a letter inside the considered string.
- If $a \neq b, m = r = 1$ and $z \neq \epsilon$ then ba is listed among the pairs and we try to compress the occurrence right after the first z. If we fail then it means that one of the letters was compressed with its neighbouring letter, which is also in the string.

Consider the first substring that is compressed and it is wholly within $zb^r(a^m zb^r)^{\ell-2}a^m z$. There are two cases: the compressed substring is a block of letters or it is a pair. We give a detailed analysis in the latter case, the analysis in the former case is similar.

So, let the first pair compressed wholly within this fragment $zb^r(a^m zb^r)^{\ell-2}a^m z$ be cd, see Fig. 1 for an illustration. We claim that all pairs cd that occurred within this fragment at the beginning of the phase are compressed at this moment. Assume for the sake of contradiction that this is not the case. So this means that one of the letters, say c, was already compressed in some other compression performed earlier. By the choice of the compressed pair (i.e. cd), this c is compressed with a letter from outside of the fragment $zb(a^m zb^r)^{\ell-2}az$, there are two possibilities:

- *c is the last letter of* $zb^r(a^m zb^r)^{\ell-2}a^m z$: Observe that the letter succeeding c is either b or a letter representing a compressed pair/string. In the latter case we do not make a further compression, so it has to be b. This is a contradiction: each c that is a last letter of z was initially followed by b, and so in fact some compression of cb (note that by our choice the last letter of z was not b, and so $b \neq c$) was performed wholly within $zb^r(a^m zb^r)^{\ell-2}a^m z$ and it was done earlier than the compression of cd, contradiction with the choice of cd.
- *c is the first letter of* $zb^r(a^m zb^r)^{\ell-2}a^m z$: The argument is symmetric, with a preceding c in this case.

There are at least $\ell - 1$ occurrences of cd that are separated by $|w_2| - 2$ letters, i.e. the $(cdz')^{\ell-2}cd$ is a substring of $zb^r(a^m zb^r)^{\ell-2}a^m z$, for some z' of length $|w_2| - 2$, see Fig. 1. We take $w'_2 = cdz'$ and let w'_1 be the w_1 concatenated with string preceding the $(cdz')^{\ell-2}cd$ and w'_3 the w_3 concatenated with the string following this $(cdz')^{\ell-2}$ (note that the latter includes the ending cd, see Fig. 1). Clearly $|w'_2| = |w_2|$ and consequently $|w'_1| + |w'_3| = |w_1| + |w_3| + 2|w_2|$. Note that each w'_2 begins with cd,

which is the first substring even partially within w_2' that is compressed, furthermore, each of those w_2' is also followed by cd. So the compression inside each w_2' is done independently (because by the choice of cd there was no prior compression applied in w_2').

Concerning the analysis when the first compressed substring is some c^m it can be routinely verified that there are no essential differences in the analysis. □

The consequence of Lemma 15 is that when an almost periodic string is a substring of S, then we can give bounds on the period size and side size on the corresponding word after one phase of OneVarWordEq.

Lemma 16 *Consider an almost periodic substring w of $S(X)$ with period size p and side size s at the beginning of the phase. Then the corresponding substring after the phase of* OneVarWordEq *has a factorisation with period size at most $\frac{3}{4}p$ and side size at most $\frac{2}{3}s + \frac{7}{3}p$.*

There are two remarks: firstly, if period size of the original word was 1 then the given bound is $\frac{3}{4} < 1$, which holds, i.e. the corresponding word has no periodic part in the factorisation. Secondly, the first (last) letter of the substring may be compressed with the letter to the left (right, respectively), so outside of the considered substring. In such a case we still include the letter representing the replaced pair or block in the corresponding substring.

Proof Let us fix the factorisation $w_1 w_2^\ell w_3$ of w, where $p = |w_2|$ is the period size and $s = |w_1 w_3|$ is the side size. First of all, consider the special case, in which w_2 is a block of letters, say a (note that this simple borderline case is not covered by Lemma 15). Then without loss of generality we may assume that w_2 is a single letter: otherwise we replace w_2^ℓ with $a^{\ell \cdot |w_2|}$, which decreases the period size. Also without loss of generality we also may assume that w_1 does not end and w_3 does not begin with a, as otherwise we can move those letters to w_2^ℓ, decreasing the side size and not increasing the period size. Then during the block compression the $w_2^\ell = a^\ell$ is going to be replaced by a single letter (this block may also include some letters from outside of w, when w_1 or w_3 is empty, this does not affect the analysis). Now consider w_1 and its letters that are not compressed with at least one other letter inside w_1, i.e. uncompressed or compressed with letters outside w_1. For the latter, only the first letter of w_1 can be compressed with a letter outside w_1, as the last letter of w_1 is succeeded by w_2 that is a block of a and w_1 does not end with a. Concerning uncompressed letters, by Lemma 9, they are always followed by two (or more) compressed letters, those two letters are also inside w_1, except perhaps the case when this uncompressed letter is the last letter of w_1 (by a similar observation as before, the second-last letter of w_1 cannot be uncompressed, as then the last letter would have to be compressed with w_2). So the number of letters that are not compressed inside w_1 is at most 1 (the first letter) $+1$ (the last letter) $+\frac{|w_1|-2}{3}$ (other uncompressed letters). So the number of letters compressed inside w_1 is at least $|w_1| - 2 - \frac{|w_1|-2}{3} = \frac{2|w_1|-4}{3}$. During the compression at least half of them, i.e. $\frac{|w_1|-2}{3}$, is removed, so in the end the obtained word has length at most $|w_1'| \leq \frac{2|w_1|+2}{3}$. Similarly for w_3' its length is at most $\frac{2|w_3|+2}{3}$.

Adding 1 for the letter replacing w_2^ℓ we obtain $|w_1' w_2' w_3'| \leq \frac{2|w_1 w_3|+7}{3} = \frac{2s}{3} + \frac{7p}{3}$, as claimed.

In other cases, by Lemma 15 we can refactor w into $u_1 u_2^{\ell'} u_3$ such that $|u_1 u_3| \leq |w_1 w_3| + 2|w_2|$ and $|u_2| = |w_2|$ and each u_2 is compressed independently (note that $|u_2| \geq 2$). Then after one phase of OneVarWordEq the corresponding word w' can be represented as $u_1' u_2'^{\ell'} u_3'$. Let us inspect its compression rate. The argument for u_1 and u_3 is the same as for w_1 and w_3 in the previous case (note that u_2 is compressed independently, i.e. the same as w_2 in the previous case), so $|u_1'| \leq \frac{2|u_1|+2}{3}$ and $|u_3'| \leq \frac{2|u_3|+2}{3}$. As $|u_1 u_3| \leq |w_1 w_3| + 2|w_2|$, the new side size is at most $\frac{2}{3}s + \frac{4}{3}p + \frac{4}{3} \leq \frac{2}{3}s + 2p$, as $p \geq 2$. For the period size, consider u_2 and recall that it is compressed independently. By Lemma 9, each uncompressed letter (perhaps except the last one) is followed by two compressed ones, so there are at most $1 + \frac{|u_2|-2}{3} = \frac{|u_2|+1}{3}$ uncompressed letters, so at least $\frac{2|u_2|+2}{3}$ compressed ones. During the compression at least half of them is removed, and so $|u_2'| \leq |u_2| - \frac{|u_2|-1}{3} = \frac{2|u_2|+1}{3}$. For $|u_2| \geq 4$ this yields the desired compression rate $\frac{3}{4}$, for $|u_2| = 2$ and $|u_2| = 3$ observe that by Lemma 9 at least one letter inside u_2 is compressed and we know that the compressions inside u_2 are done independently, so $|u_2'| \leq |u_2| - 1$, which yields the desired bound for those two border cases. □

Imagine now we want to make a similar refactoring also for the small word (in order to draw conclusions about shortening of $S(X)$, which is small). So take $w^k v$ where both w and v are almost periodic words (with some period sizes and side sizes) and k is some number. When we look at w^k, each single w can be refactored so that its periodic part is compressed independently. Note that this is the same word, i.e. we are still given $w^k v$, though we have in mind a different factorisation of w. However, the compression of the w is influenced by the neighbouring letters, so while each of the middle w in w^{k-2} is compressed in the same way, both the first and the last w can be compressed differently. Hence, after the compression we obtain something of the form $w_1 w'^{k-2} w_2 v'$, where w_1, w', w_2, v' are almost periodic. In the next phase the process continues and we accumulate almost periodic words on both sides of $w'^{k'}$. So in general we deal with a word of the form $u w^k v$, where w is almost periodic and u, v are concatenations of almost periodic words. The good news is that we can bound the sum of side sizes and period sizes of almost periodic words occurring in u, v. Moreover, the period size of w drops by a constant factor in each phase, so after $\mathcal{O}(1)$ phases it is reduced to 0, i.e. w^k is almost periodic.

As a first technical step we show that Lemma 15 can be used to analyse what happens with a concatenation of almost periodic words in one phase of OneVarWordEq: as in the case of a single word, see Lemma 16, the sum of period sizes drops by a constant factor, while the sum of side sizes drops by a constant factor but it increases by a magnitude of sum of period sizes.

Lemma 17 *Let u, a substring of $S(X)$ at the beginning of the phase, be a concatenation of almost periodic words with a factorisation for which the sum of period sizes is p and side sizes is s. Then after one phase of* OneVarWordEq *the corresponding string u' is a concatenation of almost periodic words with a factorisation for which the sum of period sizes is at most $\frac{3}{4}p$ and sum of side sizes is at most $\frac{2}{3}s + \frac{7}{3}p$.*

Note that as in the case of Lemma 16 when sum of the period sizes is 1, then after one phase we are guaranteed that all almost periodic words in the factorisation have empty periodic parts. Moreover, as in the case of Lemma 16, the first and last letter of u may be compressed with the letters outside u, in which case we include in the corresponding word the letters that are obtained in this way.

Proof Let the promised factorisation of u into almost periodic words be $u_1 \cdot u_2 \cdots u_m$. We apply Lemma 16 to each of them. By a simple summation of the guarantees from Lemma 16 the bound on the period sizes is $\frac{3}{4}p$ while the bound on the sum of the side sizes is $\frac{2}{3}s + \frac{7}{3}p$. □

The following lemma is the crowning stone of our considerations. It gives bounds on the period sizes and side sizes for the word that can be represented as $uw^k v$, where w is almost periodic and u, v are concatenations of almost periodic words.

Lemma 18 *Suppose that at the beginning of the phase of* OneVarWordEq *a substring of $S(\mathcal{A}_i)$ can be represented as $uw^k v$, where w is almost periodic with period size $p_w > 0$ and side size s_w while u, v are concatenations of almost periodic words, let the sum of their period sizes be p_{uv} and side sizes s_{uv}. Then the corresponding substring at the end of the phase can be represented as $u'(w')^{k'}v'$, where w' is almost periodic with period size $p_{w'} \leq \frac{3}{4}p_w$ and side size $s_{w'} \leq \frac{2}{3}s_w + \frac{11}{3}p_w$ and u', v' are concatenations of almost periodic words, the sum of their period sizes is $p_{u'v'} \leq \frac{3}{4}(p_{uv} + 2p_w)$ and the sum of their side sizes $s_{u'v'}$ at most $\frac{2}{3}(s_{uv} + s_w) + \frac{7}{3}(p_{uv} + 2p_w)$.*

Proof Consider the factorisation of w as an almost periodic word. Consider first the main case, in which the periodic part of w is not a block of single letter. Then we can apply Lemma 15 to each w, obtaining a factorisation $w = w_1 w_2^\ell w_3$ such that $|w_2| \leq p_w$ and $|w_1 w_3| \leq s_w + 2p_w$. Then $uw^k v$ can be represented as

$$u\left(w_1 w_2^\ell w_3\right)^k v = uw_1\left(w_2^\ell w_3 w_1\right)^{k-1} w_2^\ell w_3 v.$$

Define $u' = uw_1$ and $v' = (w_2^\ell)w_3 v$, they are concatenations of almost periodic words, the sum of their period sizes is $p_{uv} + |w_2| \leq p_{uv} + p_v$ while side sizes $s_{uv} + |w_1| + |w_3| \leq s_{uv} + s_w + 2p_w$. Define also $w' = w_2^\ell w_3 w_1$, observe that each such w' is delimited by w_2 (it includes it in the left end and to the right there is a copy of it which is not inside this w') and each w_2 is compressed independently, so also each w' is compressed independently, so in particular it is compressed in the same way. Thus $u'w'^{k-1}v'$ is compressed into $u''w''^{k-1}v''$, let us estimate their sizes.

For u' and v' can can straightforwardly apply Lemma 17, obtaining that the sum of period sizes is at most $\frac{3}{4}(p_{uv} + p_w)$ while their side sizes $\frac{2}{3}(s_{uv} + s_w + 2p_w) + \frac{7}{3}(p_{uv} + p_w) = \frac{2}{3}(s_{uv} + s_w) + \frac{7}{3}p_{uv} + \frac{11}{3}p_w$. Concerning w': we apply Lemma 16, which shows that the new period size is at most $\frac{3}{4}p_w$ and new side size $\frac{2}{3}(s_w + 2p_w) + \frac{7}{3}p_w = \frac{2}{3}s_w + \frac{11}{3}p_w$, so all as claimed.

Let us return to the trivial case, in which $w = w_1 w_2^\ell w_3$ and w_2 is a block of a single letter. Note that without a loss of generality, we can assume that w_2 is a single letter

(we can replace w_2^ℓ with $a^{|w_2|\ell}$) and that w_1 does not end and w_3 does not begin with a (we can move those letters to w_2, decreasing side size and not increasing the period size). Then $uw^kv = u(w_1a^\ell w_3)^k v$. If $w_1w_3 = \epsilon$ this is equal to $ua^{k\ell}v$, we treat $a^{k\ell}$ as a almost periodic word with period size 1 and side size 0, so $ua^{k\ell}v$ have a sum of period sizes $p_{uv} + 1 = p_{uv} + p_w$ and sum of side sizes s_{uv}. Applying Lemma 17 yields the claim: the sum of period sizes is at most $\frac{3}{4}(p_{uv} + p_w)$ while the new side sizes $\frac{2}{3}s_{uv} + \frac{7}{3}(p_{uv} + p_w)$. Similarly, when $k = 1$ we can treat uwv as a concatenation of almost periodic words, the sum of their period sizes is at most $p_{uv} + p_w$ and side sizes $s_{uv} + s_w$; again, applying Lemma 17 yields the claim: the sum of period sizes is at most $\frac{3}{4}(p_{uv} + p_w)$ while the new side sizes $\frac{2}{3}(s_{uv} + s_w) + \frac{7}{3}(p_{uv} + p_w)$.

So let us go back to the main case, in which $w_1w_3 \neq \epsilon$ and $k \geq 2$. Then $uw^kv = u(w_1a^\ell w_3)^k v = uw_1a^\ell w_3w_1(a^\ell w_3w_1)^{k-2}a^\ell w_3v$. As w_3w_1 is non-empty and does not end, nor begin with a, each a^ℓ in $(a^\ell w_3w_1)^{k-2}$ is compressed independently. We set $u' = uw_1a^\ell w_3w_1$, $v' = a^\ell w_3v$ and $w' = a^\ell w_3w_1$. Applying Lemma 17 to u' and w' yields that after one phase the sum of period sizes is $\frac{3}{4}(p_{uv} + 2p_w)$ while side size $\frac{2}{3}(s_{uv} + 2s_w) + \frac{7}{3}(p_{uv} + 2p_w)$. On the other hand, the period size of w'' is $\frac{3}{4}p_w$ while its side size at most $\frac{2}{3}s_w + \frac{7}{3}p_w$ □

With Lemma 18 established, we can prove Theorem 2.

Proof of Theorem 2 Consider the string $S(X)$, which is small. We show that within $\mathcal{O}(\log N) = \mathcal{O}(1)$ phases this string is reduced to a single letter. This means that $S(X)$ is reported in the same time. Note that in the following phases the corresponding solution (if unreported) *is not* the corresponding string, as we also pop letters from X. However, the corresponding solution is the substring of this string, in particular, after those phases it is at most a letter.

So fix a small solution and its occurrence within $S(\mathcal{A})$. It can be represented as w^kv, where w and v are almost periodic with period and side size N. We claim that in each following phase the corresponding string can be represented as $u'w'^{k'}v'$, where u' and v' are concatenations of almost periodic words, the sum of their period sizes is at most $6N$ while side sizes $78N$. Also, w' is almost periodic with side size at most $11N$ and period size dropping by $\frac{3}{4}$ in each phase (and at most N at the beginning). This claim can be easily verified by induction on the estimations given by Lemma 18: this clearly holds at the beginning (take $w' = w$, $v' = v$, $u' = \epsilon$ and $k' = k$), which shows the induction basis. Concerning the induction step, consider the word after a phase. For the new period size of w', it drops by a factor $\frac{3}{4}$ by Lemma 18, so in particular it is at most N, its side size is at most $\frac{2}{3} \cdot 11N + \frac{11}{3} \cdot 1N = 11N$, as claimed. For the new side size and period size of u', v', by the same Lemma they are at most $\frac{2}{3} \cdot (78N + 11N) + \frac{7}{3}(6N + 2N) = 59\frac{1}{3}N + 18\frac{2}{3}N = 78N$ and $\frac{3}{4}(6N + 2N) = 6N$, as claimed; this shows induction step and thus the whole claim.

As the period size of w' drops by $\frac{3}{4}$ in each phase and initially it is N, after $\mathcal{O}(\log N)$ phases w' has period size 0. Then inside $u'w'^{k'}v'$ we treat $w'^{k'}$ as a periodic word with period size $|w'| \leq 11N$ and side size 0. Thus $u'w'^{k'}v'$ is a concatenation of almost periodic words with sum of period sizes at most $11N + 6N = 17N$ and sum of side size at most $78N$. Then, by easy induction on bounds given by Lemma 17, in the following phases the corresponding string will be a concatenation of almost periodic

strings, with sum of period sizes decreasing by $\frac{3}{4}$ in each phase (and initial value $17N$) and sum of side sizes at most $119N$: the induction basis trivialy holds. For the induction step consider the word after one phase, as it was a concatenation of almost periodic words, by Lemma 17 the sum of period sizes drops by a factor of $3/4$ and the sum of side sizes is at most $\frac{2}{3} \cdot 119N + \frac{7}{3} \cdot 17N = 79\frac{1}{3}N + 39\frac{2}{3}N = 119N$; this shows the inductive step and so the whole claim.

Since the corresponding word initially has sum of period sizes at most N and this sum drops by a factor $3/4$ in each phase, after $\mathcal{O}(\log N)$ phases this sum of is reduced to 0. Thus the whole string is of length equal to the sum of side sizes, i.e. at most $119N$, which will be reduced to a single letter within $\mathcal{O}(\log N)$ rounds, as claimed. Since $N = \mathcal{O}(1)$, we obtain the claim of the lemma. □

4.4 Storing of an Equation

To reduce the running time we store duplicates of short word only once. Recall that for each equation we store lists of pointers pointing to strings that are the explicit words in this equation. We store the long words in a natural way, i.e. each long word is represented by a separate string. The short words are stored more efficiently: if two short words in equations are equal we store only one string, to which both pointers point. In this way all identical short words are stored only once (though each of them has a separate pointer pointing to it); we call such a representation *succinct*.

We show that the compression can be performed on the succinct representation, without the need of reading the actual equation. This allows bounding the running time using the size of the succinct representation and not the equation.

We distinguish two types of short words: those that are substrings of long words (normal) and those that are not (overdue). We can charge the cost of processing the normal short words to the time of processing the long words. The overdue words can be removed from the equation after $\mathcal{O}(1)$ phases after becoming overdue, so their processing time is constant per $(\mathcal{A}, i)$-word (or $(\mathcal{B}, j)$-word).

The rest of this subsection is organised as follows:

- We first give precise details, how we store short and long words, see Sect. 4.4.1 and prove that we can perform compression using only succinct representation, see Lemma 19.
- We then define precisely the normal and overdue words, see Sect. 4.4.2 as well as show that we can identify new short and overdue words, see Lemma 21. Then we show that overdue words can be removed $\mathcal{O}(1)$ phases after becoming overdue, see Lemmas 22 and 23.
- Lastly, in Sect. 4.4.3, we show that the whole compression time, summed over all phases is $\mathcal{O}(n)$. The analysis is done separately for long words, normal short words and overdue short words.

As observed in Lemma 13, see also a comment at the beginning of Sect. 4, as soon as the first or last word becomes short, the remaining running time is linear. Thus, when such a word becomes short, we drop our succinct representation and recreate out of it the simple representation used in Sects. 2 and 3. Such a recreation takes linear time.

4.4.1 Storing Details

We give some more details about the storing: All long words are stored on two doubly-linked lists, one representing the long words on the left-hand sides and the other the long words on the right-hand sides. Those words are stored on the lists according to the initial order of the words in the input equation. Furthermore, for each long word we store additionally, whether it is a first or last word of some equation (note that a short word cannot be first or last). The short words are also organised as a list, the order on the list is irrelevant. Each short word has a list of its occurrences in the equations, the list points to the occurrences in the natural order (occurrences on the left-hand sides and on the right-hand sides are stored separately).

We say that such a representation is *succinct* and its size is the sum of lengths of words stored in it (so the sum of sizes of long words, perhaps with multiplicities, plus the sum of sizes of different short words). Note that we do *not* include the number of pointers from occurrences of short words. We later show that in this way we do not need to actually read the whole equation in order to compress it; it is enough to read the words in the succinct representation, see Lemma 20.

We now show that such a storage makes sense, i.e. that if two short words become equal, they remain equal in the following phases (note again that none of them are first, nor last).

Lemma 19 *Consider any explicit words A and B in the input equation. Suppose that during* OneVarWordEq *they were transformed to $A' = B'$, none of which is a first or last word in one of the equations. Then $A = B$ if and only if $A' = B'$.*

Proof By induction on operation performed by OneVarWordEq. Since none of the A', B' is the first or last word in the equation, it means that during the whole OneVarWordEq they had X to the left and to the right. So whenever a letter was left-popped or right-popped from X, it was prepended or appended to both A and B; the same applies to cutting prefixes and suffixes. Compression is never applied to a crossing pair or a crossing block, so after it two strings are equal if and only if they were before the operation. The removal of letters [in order to preserve (1)] is applied only to first and last words, so it does not apply to words considered here. Partitioning the equation into subequations does not affect the equality of explicit words. □

We now show the main property of succinct representation: the compression (both pair and block) can be performed on succinct representation in linear time.

Lemma 20 *The compression in one phase of* OneVarWordEq *can be performed in time linear in size of the succinct representation.*

Proof Let us recall what operations we need to perform and what changes are needed when comparing with the case of one equation, see Lemma 12 We comment on the case of pair compression, the case of blocks compression is done in a similar way.

Observe first, that from Lemma 19 it follows that if an explicit short word A occurs twice in the equations (both times not as a first, nor last word of the equation) it is changed during OneVarWordEq in the same way at both those instances. This

justifies our approach of performing the operations on the words stored in the list of short words and not separately on each occurrence in the equations.

First, we perform the preprocessing, to this end we need to know the first (a) and last (b) letter, this is done by looking at the first and last word. Then we prepend b and append a to each word, except those that are first or last (first ones get only a and last ones only b). To this end we go through the list of long words and short words and append appropriate letters, note that each word stores an information, whether it is first or last, so we always know, whether to prepend or append.

Now we need to list the pairs that occur in the equation, again, this is done by going through the list. As each pair occurs in one of the words, the total size is proportional to the size of the succinct representation. Sorting then also is done in linear time (note that the size of the alphabet is at most the size of the succinct representation: each letter needs to occur somewhere).

To establish the crossing pair, it is enough to look at $A_i X A_{i+1}$, where A_i is any of the first words, after establishing this we filter out the crossing pair by going through the sorted list. Lastly, we perform the compression, using pointers to localise the occurrences of ab to be replaced. The compression of crossing pairs is done while reading the whole succinct representation, so also in linear time. □

4.4.2 Normal and Overdue Short Words

The short words stored in the tables are of two types: normal and overdue. The *normal* words are substrings of the long words or A_0^2 and consequently the sum of their sizes is proportional to the size of the long words. A word becomes *overdue* if at the beginning of the phase it is not a substring of a long word nor A_0^2. It might be that it becomes a substring of such a word later, it does not stop to be an overdue word in such a case.

Since the normal words are of size $\mathcal{O}(N) = \mathcal{O}(1)$, the sum of lengths of normal words stored in short word list is at most $\mathcal{O}(1)$ larger than the sum of sizes of the long words. Hence the processing time of normal short words can be charged to the long words. For the overdue words the analysis is different: we show that after $\mathcal{O}(1)$ phases we can remove them from the equation (splitting the equations). Thus their processing time is $\mathcal{O}(1)$ per $(\mathcal{A}, i)$-word(or $(\mathcal{B}, j)$-word), so summed over all words it yields $\mathcal{O}(n_{\mathcal{A}} + n_{\mathcal{B}}) = \mathcal{O}(n)$ in total.

The new overdue words can be identified in linear time: this is done by constructing a suffix array for a concatenation of long and short words occurring in the equations.

Lemma 21 *In time proportional to the size of succinct representation size we can identify the new overdue words.*

Proof Consider all long words $A_0, \ldots, A_m$ (with or without multiplicities, it does not matter) and all short (not yet overdue) words $A'_1, \ldots A'_{m'}$, without multiplicities; in both cases this is just a listing of words stored in the representation (except for old overdue words). We construct a suffix array for the string

$$A_0^2\$A_1\$\ldots A_m\$A'_1\$\ldots A'_{m'}\#.$$

As it was already observed that the size of the alphabet is linear in the size of the succinct representation, inside the proof of Lemma 20, the construction of the suffix array can be done in linear time [11].

Now A'_i is a factor in some A_j (the case of A_0^2 is similar, it is omitted to streamline the presentation) if and only if for some suffix A''_j of A_j the strings $A''_j\$A_{j+1}\ldots A_m\$A'_1\$\ldots\$A'_{m'}\#$ and $A'_i\$\ldots\$A'_{m'}\#$ have a common prefix of length at least $|A'_i|$. In terms of the constructed suffix array, the entries for $A'_i\$\ldots\$A'_{m'}\#$ and $A''_j\$A_{j+1}\ldots\$A_m\$A'_1\$\ldots\$A'_{m'}\#$ should have a common prefix of length at least $|A'_i|$. Recall that the length of the longest common prefix of two suffixes stored at positions $p < p'$ in the suffix array is the minimum of $LCP[p], LCP[p+1], \ldots, LCP[p'-1]$.

For fixed suffix $A'_i\$\ldots\$A'_{m'}\#$ we want to find $A''_j\$A_{j+1}\ldots\$A_m\$A'_1\$\ldots\$A'_{m'}\#$ (where A''_j is a suffix of some long word A_j) with which it has the longest common prefix. As the length of the common prefix of pth and p'th entry in a suffix array is $\min(LCP[p], LCP[p+1], \ldots, LCP[p''-1])$, this is is either the first previous or first next suffix of this form in the suffix array. Thus the appropriate computation can be done in linear time: we first go down in the suffix array, storing the last spotted entry corresponding to a suffix of some long A_j, calculating the LCP with consecutive suffixes and storing them for the suffixes of the form $A'_i\$\ldots\$A'_{m'}\#$. We then do the same going from the bottom of the suffix array. Lastly, we choose the larger from two stored values; for $A'_i\$\ldots\$A'_{m'}\#$ it is smaller than $|A'_i|$ if and only if A'_i just became an overdue word.

Concerning the running time, it linearly depends on the size of the succinct representation and alphabet size, which is also linear in size of succinct representation, as claimed. □

The main property of the overdue words is that they can be removed from the equations in $\mathcal{O}(1)$ phases after becoming overdue. This is shown by a series of lemmata.

First we need to define what does it mean that for solution word A in one side of the equation is at the same position as its copy on the other side of the equation: we say that for a substitution S the explicit word A_i (or its subword) is *arranged against* the explicit word B_j ($S(X)$ for some fixed occurrence of X) if the position within $S(\mathcal{A}_k)$ occupied by this explicit word A_i (or its subword) are within the positions occupied by explicit word B_j ($S(X)$, respectively) in $\mathcal{B}_k$.

Lemma 22 *Consider a short word A in a phase in which it becomes overdue. Then for each solution $S(X)$ either S is small or in every $S(\mathcal{A}_k) = S(\mathcal{B}_k)$ each explicit word A_i equal to A is arranged against another explicit word B_j equal to A.*

Proof Consider an equation and a solution S such that in some $S(\mathcal{A}_k) = S(\mathcal{B}_k)$ an explicit word A_i (equal to an overdue word A) is not arranged against another explicit word equal to A. There are three cases:

A is arranged against $S(X)$ Note that in this case A is a substring of $S(X)$. Either $S(X)$ is a substring of A_0 or $S(X) = A_0^k A'_0$, where A'_0 is a prefix of A_0. In the former case A is a factor of A_0, which is a contradiction, in the latter it is a factor of A_0^{k+1}.

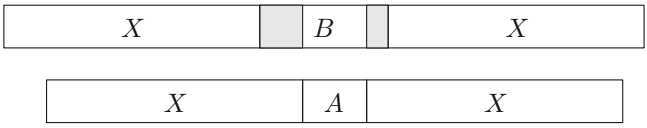

Fig. 2 A is arranged against B. The periods of length at most $|B| - |A|$ are in *ligther grey*. Since $A \neq B$, at least one of them is non-empty

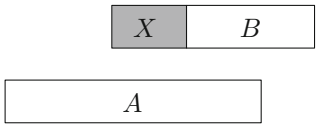

Fig. 3 Subword of A_i is arranged against the whole $S(X)$

As A_0 is long and A short, it follows that $|A| < |A_0|$ and so A is a factor of A_0^2, contradiction with the assumption that A is overdue.

A is arranged against some word Since A is an overdue word, this means that A_i is arranged against a short word B_j. Note that both A_i and B_j are preceded and succeeded by $S(X)$, since $A_i \neq B_j$ we conclude that $S(X)$ has a period at most $|B_j| - |A_i|$, see Fig. 2; in particular S is small.

Other case Since A_i is not arranged against any word, nor arranged against $S(X)$, it means that some substring of A_i is arranged against $S(X)$ and as A_i is preceded and succeeded by $S(X)$, this means that either $S(X)$ is shorter than A_i or it has a period at most $|A_i|$, see Figs. 3 and 4, respectively. In both cases S is small.

□

Observe that due to Theorem 2 and Lemma 22 the $(\mathcal{A}, i)$-words and $(\mathcal{B}, j)$-words that are overdue can be removed in $\mathcal{O}(1)$ phases after becoming overdue: suppose that A becomes an overdue word in phase ℓ. Any solution, in which an overdue word A is not arranged against another occurrence of A is small and so it is reported after $\mathcal{O}(1)$ phases. Consider an equation $\mathcal{A}_i = \mathcal{B}_i$ in which A occurs. Then the first occurrence of A in $\mathcal{A}_i$ and the first occurrence of A in $\mathcal{B}_j$ are arranged against each other for each solution S. In particular, we can write $\mathcal{A}_i = \mathcal{B}_i$ as $\mathcal{A}'_i X A X \mathcal{A}''_i = \mathcal{B}'_i X A X \mathcal{B}''_i$, where $\mathcal{A}_i$ and $\mathcal{B}_i$ do not have A as an explicit word (recall that A is not the first, nor the last word in $\mathcal{A}_i = \mathcal{B}_i$). This equation is equivalent to two equations $\mathcal{A}'_i = \mathcal{B}'_i$ and $\mathcal{A}''_i = \mathcal{B}''_i$. This procedure can be applied recursively to $\mathcal{A}''_i = \mathcal{B}''_i$. In this way, all occurrences of A are removed and no solutions are lost in the process. There may be many overdue strings so the process is a little more complicated, however, as each word can be removed once during the whole algorithm, in total it takes $\mathcal{O}(n)$ time.

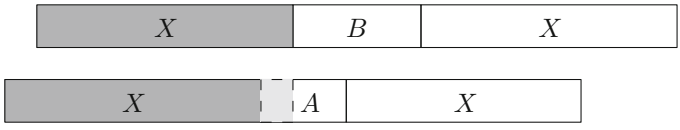

Fig. 4 Subword of A_i is arranged against $S(X)$. The overlapping $S(X)$ are in in *grey*, the $S(X)$ has a period shorter than A_i, the period is depicted in *lighter grey*

Lemma 23 *Consider the set of overdue words introduced in phase ℓ. Then in phase $\ell + \mathcal{O}(1)$ we can remove all occurrences of these overdue words from the equations.*

The obtained set of equations has the same set of solutions. The amortised time spend on removal of overdue words, over the whole run of OneVarWordEq*, is $\mathcal{O}(\#_X)$.*

Proof Consider any word A that become overdue in phase ℓ and any solution S of this equation, such that in some $S(\mathcal{A}_i) = S(\mathcal{B}_i)$ the explicit word A is not arranged against another instance of the same explicit word. Then due to Lemma 22 the $S(X)$ is small. Consequently, from Theorem 2 this solution is reported before phase $\ell + c$, for some constant c. So any solution S' in phase $\ell + c$ corresponds to a solution S from phase ℓ that had each explicit word A arranged in each $S(\mathcal{A}_i) = S(\mathcal{B}_i)$ against another explicit word A. Since all operations in a phase either transform solution, implement the pair compression of implement the blocks compression for a solution $S(X)$, it follows that in phase $\ell + c$ the corresponding overdue words A' are arranged against each other in $S'(\mathcal{A}'_i) = S'(\mathcal{B}'_i)$. Moreover, by Lemma 19 each explicit word A' in this phase corresponds to an explicit word A in phase ℓ, i.e. there are no 'new' copies of A' (recall that the fisrt and last words are long).

This observation allows removing all overdue words introduced in phase ℓ. Let $C_1, C_2, \ldots, C_m$ (in phase $\ell + c$) correspond to all overdue words introduced in phase ℓ. By Lemma 21 we have already identified the overdue words. Using the list of short words, for each overdue word C, we have the list of pointers to occurrences of C in left-hand sides of the equations and right-hand sides of the equations, those lists are sorted according to the order of occurrences. In phase $\ell + c$ we go through those lists, if the first occurrences of A in the left-hand sides and right-hand sides are in different equations then the equations are not satisfiable, as this would contradict that in each solution both A is arranged against its copy. Otherwise, they are in the same equation $\mathcal{A}_i = \mathcal{B}_i$, which is of the form $\mathcal{A}'_i X A X \mathcal{A}''_i = \mathcal{B}'_i X A X \mathcal{B}''_i$, where $\mathcal{A}'_i$ and $\mathcal{B}'_i$ do not have any occurrence of A within them. We split $\mathcal{A}_i = \mathcal{B}_i$ into two equations $\mathcal{A}'_i = \mathcal{B}'_i$ and $\mathcal{A}''_i = \mathcal{B}''_i$ and we trim them so that they are in the form described in (1). Clearly each solution of the new system of equation is also a solution of the old system, on the other hand, in any solution of the old system the copies of A were arranged against its copy, so the solution also satisfies the created equations.

Note that as new equations are created, we need to reorganise the pointers from the first/last words in the equations, however, this is easily done in $\mathcal{O}(1)$ time. The overall cost can be charge to the removed X, which makes in total at most $\mathcal{O}(\#_X)$ cost. □

4.4.3 Compression Running Time

Lemma 24 *The running time of* OneVarWordEq*, except for time used to test the solutions, is $\mathcal{O}(n)$.*

Proof By Lemma 20 the cost of compression is linear in terms of the size of the succinct representation by Lemma 21 in the same time bounds we can also identify the overdue words. Lastly, by Lemma 22 the total cost of removing the overdue words is $\mathcal{O}(n)$. So it is enough to show that the sum of sizes of the succinct representations summed over all phases is $\mathcal{O}(n)$.

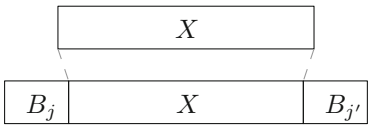

Fig. 5 Let B_j and $B_{j'}$ both have their letters arranged against letters from fixed occurrence of X. Then the X separating them is a proper substring of another X, contradiction

When the overdue words are excluded, the size of the succinct representation is proportional to the total length of long words. Since by Lemma 10 this sum of lengths decreases by a constant in each phase, the sum of those costs is linear in n.

Concerning the costs related to the overdue words: Note that an $(\mathcal{A}, i)$-word or $(\mathcal{B}, j)$-word is overdue for only $\mathcal{O}(1)$ phases, after which it is deleted from the equation see Lemma 23. So in $\mathcal{O}(1)$ phases it is charged $\mathcal{O}(N) = \mathcal{O}(1)$ cost, during the whole run of OneVarWordEq. Summing over all $(\mathcal{A}, i)$-words and $(\mathcal{A}, i)$-words yields $\mathcal{O}(n)$ time. □

4.5 Testing

We already know that thanks to appropriate storing the compression of the equations can be performed in linear time. It remains to explain how to test the solutions fast, i.e. how to perform TestSimpleSolution when all first and last words are still long.

Recall that TestSimpleSolution checks whether S,which is of the form $S(X) = a^\ell$ for some ℓ, is a solution by comparing $S(\mathcal{A}_i)$ and $S(\mathcal{B}_i)$ letter by letter, replacing X with a^ℓ on the fly. We say that in such a case a letter b in $S(\mathcal{A}_i)$ is *tested against* the corresponding letter in $S(\mathcal{B}_i)$. Note that during the testing we do not take advantage of the smaller size of the succinct representation, so we need to make a separate analysis. Consider two letters, from A_i and B_j, that are tested against each other. If one of A_i and B_j is long, this can be amortised against the length of the long word. The same applies when one of the words A_{i+1} or B_{j+1} is long. So the only problematic case is when all of those words are short. To deal with this case efficiently we distinguish between different test types, in which we exploit different properties of the solutions to speed up the tests. In the end, we show that the total time spent on testing is linear.

For a substitution S by a *mismatch* we denote the first position on which S is shown not be a solution, i.e. sides of the equation have different letters (we use a natural order on the equations); clearly, a solution has no mismatch. Furthermore, OneVarWordEq stops the testing as soon as it finds a mismatch, so in the rest of this section, if we use a name *test* for a comparison of letters, this means that the compared letters are before the mismatch (or that there is no mismatch at all).

There are two preliminary technical remarks: First we note that when testing a substitution S, for a fixed occurrence of X there is at most one explicit word whose letters are tested against letters from this occurrence of X.

Lemma 25 *Fix a tested substitution S and an occurrence of X in the equation. Then there is at most one explicit word whose letters are arranged against letters from this fixed occurrence of $S(X)$.*

Proof Without loss of generality assume that X occurs within $\mathcal{A}_\ell$ in an equation $\mathcal{A}_\ell = \mathcal{B}_\ell$. Suppose that B_j and $B_{j'}$ (for $j' > j$) have their letters arranged against a letter from this fixed occurrence of $S(X)$, see Fig 5. But B_j and $B_{j'}$ are separated by at least one X in the equation, and whole this X is also arranged against this fixed occurrence of X, contradiction. □

As a second remark, observe that tests include not only explicit letters from $S(\mathcal{A}_\ell)$ and $S(\mathcal{B}_\ell)$ but also letters from $S(X)$. In the following we will focus on tests in which at least one letter comes from an explicit word. It is easy to show that the time spent on other tests is at most as large as time spent on those tests. This follows from the fact that such other tests boil down to comparison of long blocks of a and the previous test is of a different type, so we can account the comparison between two long blocks of a to the previous test. However, our fast testing procedures in some times makes a series of tests in $\mathcal{O}(1)$ time, so this argument can be made precise only after the explanation of the details of various testing optimisations. For this reason the proof of Lemma 26 is delayed till the end of this section.

Lemma 26 *Suppose that we can perform all tests in which at least one letter comes from an explicit word in $\mathcal{O}(n)$ time. Then we can perform all test in $\mathcal{O}(n)$ time.*

Thus, in the following of this section we consider only the tests in which at least one letter comes from an explicit word.

4.5.1 Test Types

Suppose that for a substitution S a letter from A_i is tested against a letter from $S(XB_j)$ or a letter from B_j is tested against a letter from $S(XA_i)$ (the special case, when there is no explicit word after X is explained later). We say that this test is:

- *protected:* if at least one of A_i, A_{i+1}, B_j, B_{j+1} is long;
- *failed:* if A_i, A_{i+1}, B_j and B_{j+1} are short and a mismatch for S is found till the end of A_{i+1} or B_{j+1};
- *aligned:* if $A_i = B_j$ and $A_{i+1} = B_{j+1}$, all of them are short and the first letter of A_i is tested against the first letter of B_j;
- *misaligned:* if all of A_i, A_{i+1}, B_j, B_{j+1} are short, $A_{i+1} \neq A_i$ or $B_{j+1} \neq B_j$ and this is not an aligned nor failed test;
- *periodical:* if $A_{i+1} = A_i$, $B_{j+1} = B_j$, all of them are short and this is not an aligned nor failed test.

So far this classification does not apply to the case, when a letter from A_i is tested against letter from X that is not followed by an explicit word. There are two cases:

- If A_i is not followed by X in the equation then A_i is a last word, in particular it is long. Therefore this test is protected.
- If A_i is followed by X then there is a mismatch till the end of A_iX, so this test is failed.

Observe that 'failed test' does not mean a mismatch, just a fact that soon there will be a mismatch. The protected, misaligned and failed tests are done in a letter-by-letter

way, while the aligned and periodical tests are made in larger groups (in $\mathcal{O}(1)$ time per group, this of course means that we use some additional data structures).

It is easy to show that there are no other tests, see Lemma 27. We separately calculate the cost of each type of tests. As some tests are done in groups, we distinguish between number of tests of a particular type (which is the number of letter-to-letter comparisons) and the time spent on test of a particular type (which may be smaller, as group of tests are performed in $\mathcal{O}(1)$ time); the latter includes also the time needed to create and sustain the appropriate data structures.

For failed tests note that they take constant time per phase and we know that there are $\mathcal{O}(\log n)$ phases. For protected tests, we charge the cost of the protected test to the long word and only $\mathcal{O}(|C|)$ such tests can be charged to one long word C in a phase. On the other hand, each long word is shortened by a constant factor in a phase, see Lemma 10, and so this cost can be charged to those removed letters and thus the total cost of those tests (over the whole run of OneVarWordEq) is $\mathcal{O}(n)$.

In case of the misaligned tests, it can be shown that S in this case is small and that it is tested at the latest $\mathcal{O}(1)$ phases after the last of A_i, A_{i+1}, B_i, B_{i+1} becomes short, so this cost can be charged to, say, B_i becoming short and only $\mathcal{O}(1)$ such tests are charged to this B_i (over the whole run of the algorithm). Hence the total time of such tests is $\mathcal{O}(n)$.

For the aligned tests, consider the consecutive aligned tests, they correspond to comparison of $A_i X A_{i+1} \dots A_{i+k} X$ and $B_j X B_{j+1} \dots B_{j+k} X$, where $A_{i+\ell} = B_{j+\ell}$ for $\ell = 1, \dots, k$. So to perform them efficiently, it is enough to identify the maximal (syntactically) equal substrings of the equation and from Lemma 19 it follows that this corresponds to the (syntactical) equality of substrings in the original equation. Such an equality can be tested in $\mathcal{O}(1)$ using a suffix array constructed for the input equation (and general lcp queries on it). To bound the total running time it is enough to notice that the previous test is either misaligned or protected. There are $\mathcal{O}(n)$ such tests in total, so the time spent on aligned tests is also linear.

For the periodical test suppose that we are to test the equality of (suffix of) $S((A_i X)^\ell)$ and (prefix of) $S(X(B_j X)^k)$. If $|A_i| = |B_j|$ then the test for A_{i+1} and B_{j+1} is the same as for A_i and B_j and so can be skipped. If $|A_i| > |B_j|$ then the common part of $S((A_i X)^\ell)$ and $S(X(B_j X)^k)$ have periods $|S(A_i X)|$ and $|S(B_j X)|$ and consequently has a period $|A_i| - |B_j| \leq N$. So it is enough to test first common $|A_i| - |B_j|$ letters and check whether $|S(A_i X)|$ and $|S(B_j X)|$ have period $|A_i| - |B_j|$, which can be checked in $\mathcal{O}(1)$ time.

This yields that the total time of testing is linear. The details are given in the next subsections.

We begin with showing that indeed each test is either failed, protected, aligned, misaligned or periodical.

Lemma 27 *Each test is either failed, protected, misaligned, aligned or periodical. Additionally, whenever a test in made, in $\mathcal{O}(1)$ time we can establish, what type of test this is.*

Proof Without loss of generality, consider a test of a letter from A_i and from $S(X B_j)$. If any of A_{i+1}, B_{j+1}, A_i or B_j is long then it is protected (this includes the case in which some of A_{i+1}, B_j, B_{j+1} does not exist). Concerning the running time, for each

explicit word we keep a flag, whether it is short or long. Furthermore, as each explicit word has a link to its successor and predecessor, we can establish whether any of A_{i+1}, B_{j+1}, A_i or B_j is long in $\mathcal{O}(1)$ time.

So consider the case in which all A_{i+1}, B_{j+1}, A_i and B_j (if they exist) are short, which also can be established in $\mathcal{O}(1)$ time. It might be that this test is failed (again, some of the words A_{i+1}, B_j, B_{j+1} may not exist), too see this we need to make some look-ahead tests, but this can be done in $\mathcal{O}(N)$ time (we do not treat those look-aheads as tests, so there is not recursion here).

Otherwise, if the first letter of A_i and B_j are tested against each other and $A_i = B_j$ and $A_{i+1} = B_{j+1}$ then the test is aligned (clearly this can be established in $\mathcal{O}(1)$ time using look-aheads). Otherwise, if $A_{i+1} \neq A_i$ or $B_{j+1} \neq B_j$ then it is misaligned (again, $\mathcal{O}(1)$ time for look-aheads). In the remaining case $A_{i+1} = A_i$ and $B_{j+1} = B_j$, so this is a periodical test. □

4.5.2 Failed Tests

We show that in total there are $\mathcal{O}(\log n)$ failed tests. This follows from the fact that there are $\mathcal{O}(1)$ substitutions tested per phase and there are $\mathcal{O}(\log n)$ phases.

Lemma 28 *The number of all failed tests is* $\mathcal{O}(\log n)$ *over the whole run of* OneVarWordEq.

Proof As noticed, there are $\mathcal{O}(1)$ substitutions tested per phase. Suppose that the mismatch is for the letter from A_i and a letter from XB_j (the case of XA_i and B_j is symmetrical). Then the failed tests include at least one letter from $XA_{i-1}XA_i$ or $XB_{j-1}XB_jX$, assuming they come from a short word. There are at most $4N$ failed tests that include a letter from A_{i-1}, A_i, B_{j-1}, B_j (as if the test is failed then in particular this explicit word is short). Concerning the tests including the occurrences of X in-between them, observe that by Lemma 25 each such X can have tests with at most one short word, so this gives additional $5N$ tests. Since $N = \mathcal{O}(1)$, we conclude that there are $\mathcal{O}(1)$ failed tests per phase and so $\mathcal{O}(\log n)$ failed tests in total, as there are $\mathcal{O}(\log n)$ phases, see Lemma 10. □

4.5.3 Protected Tests

As already claimed, the total number of protected tests is linear in terms of length of long words: to show this it is enough to charge the cost of the protected test to the appropriate long word and see that a long word A can be charged only $|A|$ such tests for test including letters from A and $\mathcal{O}(1)$ letters from neighbouring short words, which yields $\mathcal{O}(|A|)$ tests. As the length of the long words drops by a constant factor, summing this up over all phases in which this explicit word is long yields $\mathcal{O}(n)$ tests in total.

Lemma 29 *In one phase the total number of protected tests is proportional to the length of the long words. In particular, there are* $\mathcal{O}(n)$ *such test during the whole run of* OneVarWordEq.

Proof As observed in Lemma 26 we can consider only tests in which at least one letter comes from an explicit word. Suppose that a letter from A_i takes part in the protected test (the argument for a letter from B_j is similar, it is given later on) and it is tested against a letter from XB_j, then one of A_i, A_{i+1}, B_j, B_{j+1} is long, we charge the cost according to this order, i.e. we charge it to A_i if it is long, if A_i is not but A_{i+1} is long, we charge it to A_{i+1}, if not then to B_j if it is long and otherwise to B_{j+1}. The analysis and charging for a test of a letter from B_j is done in a symmetrical way (note that when the test includes two explicit letters, we charge it twice, but this is not a problem).

Now, fix some long word A_i, we estimate, how many protected tests can be charged to it. It can be charged with cost of tests that include its own letters, so $|A_i|$ tests. When A_{i-1} is short, it can also charge tests in which its letters take part. As it is short, it is at most $\mathcal{O}(N) = \mathcal{O}(1)$ such tests.

Also some words from $\mathcal{B}$ can charge the cost of tests to A_i, we can count only the test in which letters from A_i do not take part. This can happen in two situations: letters tested against XA_i and letters tested against XA_{i-1} (in which case we additionally assume that A_{i-1} is short). We have already accounted the tests made against A_{i-1} and A_i and by Lemma 25 for each occurrence of X there is at most one explicit word whose letters are tested against this occurrence of X. Those that were charged to A_i come from short words, so there are additionally at most $2N$ tests of this form.

So in total A_i is charged only $\mathcal{O}(|A_i|)$ in a phase. From Lemma 10 the sum of lengths of long words drops by a constant factor in each phase, and as in the input it is at most n, the total sum of number of protected tests is $\mathcal{O}(n)$. □

4.5.4 Misaligned Tests

On the high level, in this section we want to show that if there is a misaligned test then the tested solution is small and use this fact for accounting the cost of such tests. However, this statement is trivial, as we test only solutions of the form a^k for some k, which are always small. To make this statement more meaningful, we generalise the notion of a misaligned test for arbitrary substitutions, not only the tested one. In this way two explicit words A_i and B_j can be misaligned for a substitution S. We show three properties of this notion:

- M1 If there is a misaligned test for a substitution S for a letter from A_i against letter in XB_j or a letter from B_j against letter from XA_i then A_i and B_j are misaligned for S. This is shown in Lemma 30.
- M2 If there are misaligned words A_i and B_j for a solution S then S is small, as shown in Lemma 31.
- M3 If A_i and B_j are misaligned for S in a phase ℓ then S is reported in phase ℓ or the corresponding words A'_i and B'_j in phase $\ell + 1$ are also misaligned for the corresponding S', see Lemma 32.

Those properties are enough to improve the testing procedure so that one $(\mathcal{A}, i)$-word (or $(\mathcal{B}, j)$-word) takes part in only $\mathcal{O}(1)$ misaligned tests: suppose that A_i becomes small in phase ℓ. Then all solutions, for which it is misaligned with some B_j, are small by (M2). Hence, by Theorem 2, all of those solutions are reported (in

particular: tested) within the next c phases, for some constant c. Thus, if A_i takes part in a misaligned test (for S) in phase $\ell' > \ell + c$ then S is not a solution: by (M1) A_i and appropriate B_j are misaligned and by (M3) they were misaligned also in phase ℓ (for the corresponding solution S'), and solution S' was reported before phase ℓ', by (M2). Hence we can immediately terminate the test; therefore A_i can take part in misaligned tests in phases $\ell, \ell + 1, \ldots, \ell + c$, i.e. $\mathcal{O}(1)$ ones. This plan is elaborated in this section, in particular, some technical details (omitted in the above description) are given.

We say that A_i and B_j that are blocks from two sides of one equations $\mathcal{A}_\ell = \mathcal{B}_\ell$ are *misaligned for a substitution* S if

- a mismatch for S is not found till the end of A_{i+1} or B_{j+1};
- all A_{i+1}, A_i, B_{j+1} and B_j are short;
- either $A_i \neq A_{i+1}$ or $B_j \neq B_{j+1}$;
- it does not hold that $A_i = B_j$ and $A_{i+1} = B_{j+1}$ and the first letter of A_i is at the same position as the first letter of B_j under substitution S;
- the position of the first letter of A_i in $S(\mathcal{A}_\ell)$ is among the position of $S(XB_j)$ in $S(\mathcal{B}_\ell)$ or, symmetrically, the position of the first letter of B_j in $S(\mathcal{B}_\ell)$ is among the position of $S(XA_i)$ in $S(\mathcal{A}_\ell)$.

We show (M1), which shows that the definitions of misaligned blocks and misaligned tests are reformulations of each other.

Lemma 30 *If a letter from A_i is tested (for S) against a letter from XB_j and this test is misaligned then A_i and B_j are misaligned for S; similar statement holds for letters from B_j.*

Proof This is just a reformulation of a definition (we consider only the case of letters from A_i, the argument for letters from B_j is symmetrical):

- Since this is not a failed test, there is no mismatch till the end of A_{i+1} and B_{j+1}.
- As this is not a protected test, all A_i, A_{i+1}, B_j and B_{j+1} are short.
- As this is a misaligned test, either $A_i \neq A_{i+1}$ or $B_j \neq B_{j+1}$.
- As this is not an aligned test, either $A_i \neq B_j$ or $A_{i+1} \neq B_{j+1}$ or the first letter of A_i is not at the same position as the first letter of B_j (both under S).
- By the choice of B_j, the first position of A_i under S is among the positions of XB_j (under S).

□

We move to showing (M2). It follows by considering $S(XA_iXA_{i+1}X)$ and $S(XB_jXB_{j+1}X)$. The large amount of $S(X)$ in it allows showing the periodicity of fragments of $S(X)$ and in the end, that S is small.

Lemma 31 *When the A_i and B_j are misaligned for a solution S then S is small.*

Proof Suppose that A_i and B_j are from an equation $\mathcal{A}_\ell = \mathcal{B}_\ell$. In the proof we consider only one of the symmetric cases, in which A_i is begins not later than B_j (i.e. the first letter of A_i is arranged against the letter from XB_j), the other case is shown similarly.

There are two main cases: either some of A_i, A_{i+1}, B_j and B_{j+1} has some of its letters arranged against an explicit word or all those words are arranged against (some occurrences) of X.

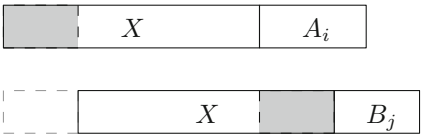

Fig. 6 A letter from B_j is arranged against the letter from A_i. The period of $S(X)$ is in *grey*

One of the Words has Some of Its Letters Arranged Against an Explicit Word

We claim that in this case S has a period of length at most N, in particular, it is small. First of all observe that it is not possible that *each* of A_i, A_{i+1}, B_j and B_{j+1} has *all* of its letters arranged against letters of an explicit word: since A_i is arranged against XB_j this would imply that A_i is arranged against B_j (in particular, their first letters are at corresponding positions) and (as no mismatch is found till end of A_i and B_j) so $A_i = B_j$. Similarly, $A_{i+1} = B_{j+1}$. This contradicts the assumption that A_i and B_j are misaligned.

Thus, there is a word among A_i, A_{i+1}, B_j and B_{j+1}, say B_j, that is partially arranged against an explicit word and partially against X (note that this explicit word does not have to be among A_i, A_{i+1}, B_j and B_{j+1}), see Fig. 6. As each explicit words is proceeded and succeeded by X, it follows that $S(X)$ has a period at most N.

All Words have all their Letters Arranged Against Occurrences of X

In the following we assume that letters from A_i, A_{i+1}, B_j and B_{j+1} are arranged against the letters from $S(X)$. Observe that due to Lemma 25 this means that whole A_i is arranged against $S(X)$ preceding B_j, the B_j against $S(X)$ preceding A_{i+1}, whole A_{i+1} against $S(X)$ preceding B_{j+1} and whole B_{j+1} against $S(X)$ succeeding A_{i+1}, see Fig. 7.

Let $a = |A_i|, b = |B_j|$ and $x = |S(X)|$, as in Fig. 7. There are three cases: $a > b$, $a < b$ and $a = b$, we consider them separately.

Consider first the case in which $a > b$, see Fig. 7. Let p denote the offset between the $S(X)$ preceding A_i and the one preceding B_j; then $S(X)$ has a period p. Similarly, when we consider the $S(X)$ succeeding A_i and the one succeeding B_j we obtain that the offset between them is $p - a + b$, which is also a period of $S(X)$. Those offsets correspond to borders (of $S(X)$) of lengths $x - p$ and $x - p + a - b$, see Fig. 7. Then the shorter border (of length $x - p$) is also a border of the longer one (of length $x - p + a - b$), hence the border of length $x - p + a - b$ has a period $a - b$, so it is of

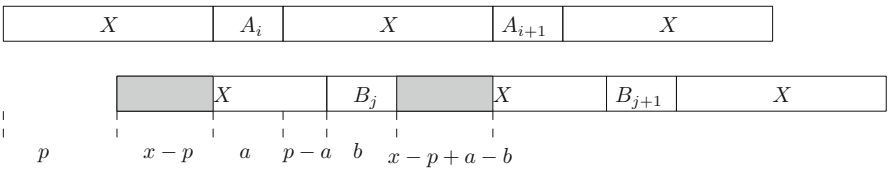

Fig. 7 The letters of A_i, A_{i+1}, B_j and B_{j+1} are arranged against the letters from $S(X)$. The lengths of fragments of text are beneath the figure, between *dashed lines*. Comparing the positions of the first and second $S(X)$ yields that p is a period of $S(X)$, second and third that $x - p + a$ while the third and fourth that $p - a + b$ is. The borders of $S(X)$ corresponding to the first and third one are marked in *grey*

the form $w^k u$, where $|w| = a - b$ and $|u| < a - b$. Now, the prefix of $S(X)$ of length $x - p + a$ is of the form $w^k u'$, for some u' of length less than a (as this is a prefix of length $x - p + a - b$ extended by the following b letters). When we compare the positions of $S(X)$ preceding B_j and the one succeeding A_i we obtain that $S(X)$ has a period $x - p + a$ so the whole $S(X)$ is of the form $(w^k u')^\ell w'$, where w' is a prefix of $w^k u'$, hence S is small: w and u' are of length at most N, as w' is a prefix of $w^k u'$, either it is a prefix of w^k, so it is of the form $w^{k'} w''$ where w'' is a prefix of w, or it includes the whole w^k, so it is of the form $w^k u''$, where u'' is a prefix of u'.

Consider the symmetric case, in which $b > a$ and again use Fig. 7. The same argument as before shows that p and $p - a + b$ are periods of $S(X)$ and the corresponding borders are of length $x - p$ and $x - p + a - b$. Now, the shorter of them (of length $x - p + a - b$) is a border of longer of them (of length $x - p$), so the prefix of length $x - p$ of $S(X)$ has a period $b - a$, so it is of the form $w^k u$, where $|w| = b - a$ and $|u| < b - a$. Hence the prefix of length $x - p + a$ is of the form $w^k u'$ for some u' of length less than b. As in the previous case, $S(X)$ has a period $x - p + a$ and so the whole $S(X)$ is of the form $(w^k u')^\ell w'$, where w' is a prefix of $w^k u'$, hence S is small.

Consider now the last case, in which $|A_i| = |B_j|$. If $|A_{i+1}| \neq |A_i|$ then $|B_j| \neq |A_{i+1}|$ and we can repeat the same argument as above, with B_j and A_{i+1} taking the roles of A_i and B_j, which shows that S is small. So consider the case in which $|A_{i+1}| = |A_i|$. If $|B_j| \neq |B_{j+1}|$ then again, repeating the argument as above for A_{i+1} and B_{j+1} yields that S is small. So we are left with the case in which $|A_{i+1}| = |A_i| = |B_j| = |B_{j+1}|$. Then A_{i+1} is arranged against the same letters in $S(X)$ as A_i and B_{j+1} is arranged against the same letters in $S(X)$ as B_j. As there is no mismatch till the end of A_{i+1} and B_{j+1}, we conclude that $A_{i+1} = A_i$ and $B_{j+1} = B_j$ contradicting the assumption that A_i and B_j are misaligned, so this case is non-existing. □

We now show that if A_i and B_j are misaligned for S then they were (for a corresponding solution) in the previous phase (assuming that all involved words were short). This is an easy consequence of the way explicit words are modified (we prepend and append the same letters and compress all explicit words in the same way).

Lemma 32 *Suppose that A_i and B_j are misaligned for a solution S. If at the previous phase all A'_{i+1}, A'_i, B'_{j+1} and B'_j were short then A'_i and B'_j were misaligned for the corresponding solution S'.*

Proof We verify the conditions on misaligned words point by point:

- Since S' is a solution, there is no mismatch.
- By the assumption, all A'_{i+1}, A'_i, B'_{j+1} and B'_j are short.
- We know that either $A_i \neq A_{i+1}$ or $B_j \neq B_{j+1}$ and so by Lemma 19 either $A'_i \neq A'_{i+1}$ or $B'_j \neq B'_{j+1}$ (observe that none of them is the last nor first, as they are not in the next phase).
- Suppose that $A'_i = B'_j$, $A'_{i+1} = B'_{j+1}$ and under S' the first letters of A'_i and B'_j are arranged against each other. By Lemma 19 it follows that $A_i = B_j$, $A_{i+1} = B_{j+1}$. Observe that left-popping and right popping preserves the fact that the first letters of $(\mathcal{A}, i)$-word and $(\mathcal{B}, j)$-word are arranged against each other for S' (as $S(\mathcal{A})$ and $S'(\mathcal{A}')$ are the same words). As S' is a solution, the same applies to pair compression and block compression. Hence, the first letters of A_i and B_j are

arranged against each other, contradiction with the assumption that A_i and B_j are misaligned.
– Suppose that the first letter of A_i is arranged against a letter from $S(XB_j)$. Consider, how A'_i and XB'_j under S' are transformed to A_i and XB_j under S. As in the above item, popping letters does not influence whether the first letter of $(\mathcal{A}, i)$-word is arranged against letter from $S(X)$ and $(\mathcal{B}, j)$-word (as $S(\mathcal{A})$ and $S'(\mathcal{A}')$ are the same words). Since S' is a solution, the same applies also to pair and block compression. So the position of the first letter of A_i is among the position of $S(XB_j)$ if and only if the first letter of A'_i is arranged against a letter from $S'(XB'_j)$.

The case in which the position of the first letter of B_j is among the position of $S(XA_i)$ is shown in a symmetrical way. □

Now we are ready to give the improved procedure for testing and estimate the number of the misaligned tests in it.

Lemma 33 *There are $\mathcal{O}(n)$ misaligned tests during the whole run of* OneVarWordEq.

Proof Consider a tested solution S and a misaligned test for a letter from A_i against a letter from XB_j (the case of test of letters from B_j tested against XA_i the argument is the same). Let ℓ be the number of the first phase, in which all $(\mathcal{A}, i)$-word, $(\mathcal{A}, i+1)$-word, $(\mathcal{B}, j)$-word and $(\mathcal{B}, j+1)$-word are short. We claim that this misaligned test happens between ℓth and $\ell + c$ phase, where c is the $\mathcal{O}(1)$ constant from Theorem 2.

Let A'_i and B'_j be the corresponding words in the phase ℓ. Using induction on Lemma 32 it follows that A'_i and B'_j are misaligned for S'. Thus by Lemma 31 the S' is small and thus by Theorem 2 it is reported till phase $\ell + c$. So it can be tested only between phases ℓ and $\ell + c$, as claimed.

This allows an improvement to the testing algorithm: whenever (say in phase ℓ) a letter from A_i has a misaligned test against a letter from $S(XB_j)$ we can check (in $\mathcal{O}(1)$ time), in which turn ℓ' the last among $(\mathcal{A}, i)$-word, $(\mathcal{A}, i+1)$-word, $(\mathcal{B}, j)$-word and $(\mathcal{B}, j+1)$-word became small (it is enough to store for each explicit word the number of phase in which it became small). If $\ell' + c < \ell$ then we can terminate the test, as we know already that S is not a solution. Otherwise, we continue.

Concerning the estimation of the cost of the misaligned tests (in the setting as above), there are two cases:

– *The misaligned tests that lead to the rejection of S*: This can happen once per tested solution and there are $\mathcal{O}(\log n)$ tested solution in total ($\mathcal{O}(1)$ per phase and there are $\mathcal{O}(\log n)$ phases).
– *Other misaligned tests:* The cost of the test (of a letter from A_i tested against $S(XB_j)$) is charged to the last one among $(\mathcal{A}, i)$-word, $(\mathcal{A}, i+1)$-word, $(\mathcal{B}, j)$-word and $(\mathcal{B}, j+1)$-word that became short. By the argument above, this means that this word became short within the last c phases.

Let us calculate, for a fixed $(\mathcal{A}, i)$-word (the argument for $(\mathcal{B}, j)$-word is symmetrical) how many misaligned tests of this kind can be charged to this word. They can be charged only within c phases after this word become short. In a fixed phase we test

only a constant (i.e. 5) substitutions. For a fixed substitution, A_i can be charged the cost of tests in which letters from A_i or A_{i-1} are involved (providing that A_i or A_{i-1} is short), which is at most $2N$. They can be charged also the tests from letters from B_j that is aligned against X preceding A_{i-1} or X preceding A_i (providing that B_j as well as A_{i-1} are short). Note that there is only one B_j whose letter are aligned against X preceding A_{i-1} and one for X preceding A_i, see Lemma 25, so when they are short this gives additional $2N$ tests.
This yields that one $(\mathcal{A}, i)$-word is charged $\mathcal{O}(N) = \mathcal{O}(1)$ tests in total. Summing over all words in the instance yields the claim of the lemma. □

4.5.5 Aligned Tests

Suppose that we make an aligned test, without loss of generality consider the first such test in a sequence of aligned tests. Let it be between the first letter of A_i and the first letter in B_j (both of those words are short). For this A_i and B_j we want to perform the whole sequence of successive aligned tests at once, which corresponds of jumping to A_{i+k} and B_{j+k} within the same equation such that

- $A_{i+\ell} = B_{i+\ell}$ for $0 \leq \ell < k$ and
- $A_{i+k} \neq B_{j+k}$ or one of them is a last word or $A_{i+k}X$ or $B_{j+k}X$ ends one side of the equation.

Note that this corresponds to a syntactical equality of fragments of the equation, which, by Lemma 19, is equivalent to a syntactical equality of fragments of the original equation. We preprocess (in $\mathcal{O}(n)$ time) the input equation (building a suffix array equipped with a structure answering general lcp queries) so that in $\mathcal{O}(1)$ we can return such k as well as the links to A_{i+k} and B_{j+k}. In this way we perform all equality tests for $A_i X A_{i+1} X \ldots A_{i+k-1} X = B_j X B_{j+1} X \ldots X B_{j+k-1} X$ in $\mathcal{O}(1)$ time.

To simplify the considerations, when $A_i X$ ($B_j X$) ends one side of the equation, we say that this A_i (B_j, respectively) is *almost last* word. Observe that in a given equation exactly one side has a last word and one an almost last word.

Lemma 34 *In $\mathcal{O}(n)$ we can build a data structure which given equal A_i and B_j in $\mathcal{O}(1)$ time returns the smallest $k \geq 1$ and links to A_{i+k} and B_{j+k} such that $A_{i+k} \neq B_{j+k}$ or one of A_{i+k}, B_{j+k} is a last word or one of A_{i+k}, B_{j+k} is an almost last word.*

Note that it might be that some of the equal words $A_{i+\ell} = B_{i+\ell}$ are long, and so their tests should be protected (also, the tests for some neighbouring words). So in this way we also make some free protected tests, but this is not a problem. Furthermore, the returned A_{i+k} and B_{j+k} are guaranteed to be in the same equation.

Proof First of all observe that for A_i and B_j it is easy to find the last word in their equation as well as the almost last word of the equation: when we begin to read a particular equation, we have the link to both the last word and the almost last word of this equation and we can keep them during the testing of this equation. We also know the numbers of those words so we can also calculate the respective candidate for k. So it is left to calculate the minimal k such that $A_{i+k} \neq A_{j+k}$.

Let A'_i, B'_j etc. denote the corresponding original words of the input equation. Observe that by Lemma 19 it holds that $A_{i+\ell}{}' = B'_{j+\ell}$ if and only if $A_{i+\ell} = B_{j+\ell}$ as long as none of them is last or first word. Hence, it is enough to be able to answer such queries for the input equation: if the returned word is in another equation then we should return the last or almost last word instead.

To this end we build a suffix array [11] for the input equation, i.e. for a string $A'_1 X A'_2 X \ldots A''_{n_\mathcal{A}} X B'_1 X B'_2 X \ldots B'_{n_\mathcal{B}}\$$. Now, the lcp query for suffixes $A'_i \ldots \$$ and $B'_j \ldots \$$ returns the length of the longest common prefix. We want to know what is the number of explicit words in the common prefix, which corresponds to the number of Xs in this common prefix. This information can be easily preprocessed and stored in the suffix array: for each position ℓ in $A'_1 X A'_2 X \ldots A'_{n_\mathcal{A}} X B'_1 X B'_2 X \ldots B'_{n_\mathcal{B}}\$$ we store, how many Xs are before it in the string and store this in the table $prefX$. Then when for a suffixes beginning at positions p and p' we get that their common prefix is of length ℓ, the $prefX[p+\ell] - prefX[p]$ is the number of Xs in the common prefix in such a case. If none of A_i, A_{i+1}, …, A_{i+k} nor B_j, B_{j+1}, …, B_{j+k} is the last word nor it ends the equation (i.e. they are all still in one equation) by Lemma 19 the k is the answer to our query (as $A_i = B_j$, $A_{i+1} = B_{j+1}$,…and $A_{i+k} \neq B_{j+k}$ and none of them is a last word, nor none of them ends the equation). To get the actual links to those words, at the beginning of the computation we make a table, which for each i returns the pointer to $(\mathcal{A}, i)$-word and $(\mathcal{B}, i)$-word. As we know i, j and k we can obtain the appropriate links in $\mathcal{O}(1)$ time. So it is left to compare the value of k with the value calculated for the last word and almost last word and choose the one with smaller k and the corresponding pointers. □

Using this data structure we perform the aligned tests is in the following way: whenever we make an aligned test (for the first letter of A_i and the first letter of B_j), we use this structure, obtain k and jump to the test of the first letter of A_{i+k} with the first letter of B_{j+k} and we proceed with testing from this place on. Concerning the cost, by easy case analysis it can be shown that the test right before the first of sequence of aligned tests (so the test for the last letters of A_{i-1} and B_{j-1}) is either protected or misaligned. There are only $\mathcal{O}(n)$ such tests (over the whole run of OneVarWordEq), so the time spend on aligned tests is $\mathcal{O}(n)$ as well.

Lemma 35 *The total cost aligned test as well as the usage of the needed data structure is* $\mathcal{O}(n)$.

Proof We formalise the discussion above. In $\mathcal{O}(1)$ we get to know that this is an aligned test, see Lemma 27. Then in $\mathcal{O}(1)$, see Lemma 34, we get the smallest k such that $A_{i+k} \neq B_{j+k}$ or one of them is an almost last word for this equation or the last word for this equation. We then jump straight to the test for the first letter of A_{i+k} and B_{j+k}.

Consider A_{i-1} and B_{j-1} we show that the test for their last letters (so the test immediately before the first aligned one) is protected or misaligned. By Lemma 27 it is enough to show that it is not aligned, nor periodic, nor failed.

- If it were failed then also the test for the first letters of A_i and B_j would be failed.
- It cannot be aligned, as we chose A_i and B_j as the first in a series of aligned tests.
- If it were periodic, then $A_{i-1} = A_i$ and $B_{j-1} = B_j$ while by assumption $A_i = B_j$, which implies that this test is in fact aligned, which was already excluded.

Hence we can associate the $\mathcal{O}(1)$ cost of whole sequence of aligned test to the previous test, which is misaligned or protected. Clearly, one misaligned or protected test can be charged with only one sequence of aligned tests (as it is the immediate previous test). By Lemmas 29 and 33 in total there are $\mathcal{O}(n)$ misaligned and protected tests. Thus in total all misaligned tests take $\mathcal{O}(n)$ time. □

4.5.6 Periodical Tests

The general approach in case of periodical tests is similar as for the aligned tests: we would like to perform all consecutive periodical tests in $\mathcal{O}(N)$ time and show that the test right before this sequence of periodic tests is either protected or misaligned. As in case of aligned tests, the crucial part is the identification of a sequence of consecutive periodical tests. To identify them quickly, we keep for each short A_i the value k such that A_{i+k} is the first word that is different from A_i or is the last word or the almost last word (in the sense as in the previous section: A_{i+k} is almost last if $A_{i+k}X$ ends the side of the equation), as well as the link to this A_{i+k}. Those are easy to calculate at the beginning of each phase. Now when we perform a periodical test for a letter from A_i, we test letters from $S((AX)^k)$ against the letters from (suffix of) $S(X(BX)^\ell)$. If $|A| = |B|$ then both strings are periodic with period $|S(AX)|$ and their equality can be tested in $\mathcal{O}(|A|)$ time. If $|A| \neq |B|$ then we retrieve the values $k_{\mathcal{A}}$ and $k_{\mathcal{B}}$ which tell us what is repetition of AX and BX. If one of them is smaller than 3 we make the test naively, in time $\mathcal{O}(|A| + |B|)$. If not, we exploit the fact that $S((BX)^\ell)$ has a period $|S(BX)|$ while $S((AX)^k)$ has a period $|S(AX)|$ and so their common fragment (if they are indeed equal) has a period $||S(AX)| - |S(BX)|| = ||A| - |B||$ (note that the outer '|' denote the absolute value). Hence we check, whether $S(AX)$ and $S(BX)$ have this period and check the common fragment of this length, which can be done in $\mathcal{O}(|A| + |B|)$ time. The converse implication holds as well: if $S(AX)$ and $S(BX)$ have period $||A| - |B||$ and the first $||A| - |B||$ tests are successful then all of them are. Concerning the overall running time, as in the case of aligned test, the test right before the first periodic test is either protected or misaligned, so as in the previous section it can be shown that the time spent on periodical tests is $\mathcal{O}(n)$ during the whole OneVarWordEq.

Lemma 36 *Performing all periodical tests and the required preprocessing takes in total $\mathcal{O}(n)$ time.*

Proof Similarly as in the case of aligned tests, see Lemma 35, we can easily keep the value k and the link to A_{i+k} such that A_{i+k} is the last or almost last word in this equation, the same applies for B_{j+k}. Hence it is left to show how to calculate for each short A_i (and B_j) the k such that A_{i+k} is the first word that is different from A_i.

At the end of the phase we list all words A_i that become short in this phase, see Lemma 19, ordered from the left to the right (this is done anyway, when we identify the new short words). Note that this takes at most the time proportional to the length of all long words from the beginning of the phase, so $\mathcal{O}(n)$ in total. Consider any A_i on this list (the argument for B_j is identical), note that

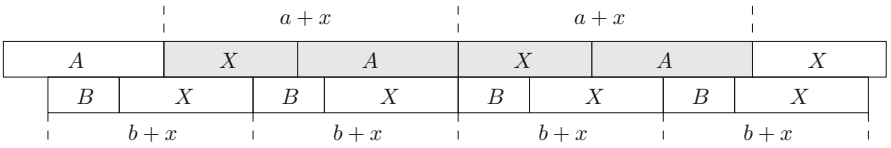

Fig. 8 The case of $a > b$. The part of $S((XA_i)^2)$ that has a period $a + x$ and $b + x$ is in *grey*

– if $A_{i+1} \neq A_i$ then A_i should store $k = 1$ and a pointer to this A_{i+1};
– if $A_i = A_{i+1}$ then A_{i+1} also became short in this phase and so it is on the list and consequently A_i should store 1 more than A_{i+1} and the same pointer as A_{i+1}.

So we read the list from the right to the left, let A_i be an element on this list. Using the above condition, we can establish in constant time the value and pointer stored by A_i. This operation is performed once per $(\mathcal{A}, i)$-word, so in total takes $\mathcal{O}(n)$ time.

Consider a periodic test, without loss of generality suppose that a letter from A_i is tested against a letter from XB_j (in particular, A_i begins not later than B_j), let the k_A and k_B be stored by A_i and B_j; as this is a periodical test, both k_A and k_B are greater than 1. Among A_{i+k_A} and B_{j+k_B} consider the one which begins earlier under substitution S: this can be determined in $\mathcal{O}(1)$ by simply comparing the lengths, the length on the A-side of the equation is $k_A(|A_i| + |S(X)|)$ while B-side length is $k_B(|B_j|+|S(X)|)+m$, where m is the remainder of $S(X)$ that is compared with A_i. Let k and ℓ be the smallest numbers such that the common part of $S(A_iX \cdots XA_{i+k-1}X)$ and $S(B_jX \cdots XB_{j+\ell-1}X)$ contain the common part of $S(A_iX \cdots XA_{i+k_A-1}X)$ and $S(B_jX \cdots XB_{j+k_B-1}X)$.

A_{i+k} and $B_{j+\ell}$ be the Note that the test for the first letter of this word is not periodic, so when we jump to it we skip the whole sequence of periodic tests. We show that in $\mathcal{O}(1)$ time we can perform the tests for all letters before this word and that the test right before the first test for A_i is protected or misaligned.

Let $a = |A_i|, b = |B_j|$ and $x = |S(X)|$. First consider the simpler case in which $a = b$. Then the tests for A_{i+1}, …, A_{i+k-1} are identical as for A_i, and so it is enough to perform just the test for A_i and B_j and then jump right to A_{i+k}.

So let us now consider the case in which $a > b$. Observe that when the whole $S((B_jX)^\ell)$ is within $S((A_iX)^3)$ then this can be tested in constant time in a naive way: the length of $S((A_iX)^3)$ is $3(a + x)$ while the length of $S((B_jX)^\ell)$ is $\ell(b + x)$. Hence $3(a + x) \geq \ell(b + x)$ and so $\ell \leq 3(a + x)/(b + x) \leq 3\max(a/b, x/x) \leq 3N$, because a/b is at most N. Thus all tests for $S((A_iX)^3)$ and $S((B_jX)^\ell)$ can be done in $\mathcal{O}(N) = \mathcal{O}(1)$ time.

So consider the remaining case, see Fig. 8 for an illustration, when $k > 3$. We claim that the tests for $S(A_iX \cdots XA_{i+k-1}X)$ and $S(B_jX \cdots XB_{j+\ell-1}X)$ are successful if and only if

– $S(A_iX)$ and $S(B_jX)$ have period $\gcd(a + x, b + x)$ *and*
– the first $\gcd(a+x,b+x)$ tests for $S(A_iX \cdots XA_{i+k-1}X)$ and $S(B_jX \cdots XB_{j+\ell-1}X)$ are successful.

$\ominus$ First $S(XA_iXA_i)$ has period $x + a$. However, it is covered with $S((B_jX)^\ell)$, so it also has period $x + b$. Since $x + a + x + b < 2x + 2a$, it follows that also the

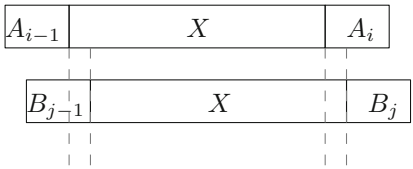

Fig. 9 The test right before the first among the sequence of periodic tests. Since A_i begins not later than B_j, B_{j-1} ends not earlier than A_{i-1}

$\gcd(x+a, x+b)$ is a period of $S(XA_iXA_i)$ and so also of $S(A_iX)$ and thus also $S(B_jX)$. The second item is obvious.

$\Leftarrow$ Since $S(A_iX)$ and $S(B_jX)$ have period $\gcd(a+x, b+x)$ also $S(A_iX\cdots XA_{i+k-1}X)$ and $S(B_jX\cdots XB_{j+\ell-1}X)$ have this period. As the first $\gcd(a+x, b+x)$ tests for $S(A_iX\cdots XA_{i+k-1}X)$ and $S(B_jX\cdots XB_{j+k-1}X)$ are successful, it follows that all the tests for their common part are.

So, to perform the test for $S(A_iX\cdots XA_{i+k-1}X)$ and $S(B_jX\cdots XB_{j+\ell-1}X)$ it is enough to: calculate $p = \gcd(a+x, b+x)$, test whether $S(A_iX)$, $S(B_jX)$ have period p and then perform the first p tests for $S(A_iX\cdots XA_{i+k-1}X)$ and $S(B_jX\cdots XB_{j+\ell-1}X)$. All of this can be done in $\mathcal{O}(1)$, since $p \le a-b \le N$ (note also that calculating p can be done in $\mathcal{O}(1)$, as $\gcd(x+a, x+b) = \gcd(a-b, x+b)$ and $a-b \le N$).

The case with $b > a$ is similar: in the special subcase we consider whether $S((A_iX)^k)$ is within $S(X(B_jX)^3)$. If so then the tests can be done in $\mathcal{O}(N)$ time. If not, then we observe that the $S(XB_{j+1}XB_{j+2})$ is covered by $S((A_iX)^k)$. So it the tests are successful, it has period both $x+b$ as well as $x+a$, so it has period $\gcd(x+a, x+b)$. The rest of the argument is identical.

For the accounting, we would like to show that the test right before the first among the considered periodic tests is not periodic. Observe, that as A_i begins not later (under S) than B_j it means that the last letter of B_{j-1} is not earlier than the last letter of A_{i-1}, see Fig. 9. So the previous test includes the last letter of B_{j-1}. It is enough to show that this test is not failed, periodic, nor aligned.

- *failed:* If it is failed then also the test for the letters in A_i are failed.
- *periodic:* If it is periodic then this contradicts our choice that the test for the first letter of A_i is the first in the sequence periodic tests.
- *aligned:* Since the first letter of A_i is arranged against XB_j, in this case the last letter of B_{j-1} needs to be arranged against the last letter of A_{i-1}. Then by the definition of the aligned test, $B_j = A_i$ and their first letters are at the same position. As by the assumption about the periodic tests we know that $A_{i+1} = A_i$ and $B_{j+1} = B_j$ we conclude that the test for the first letter of A_i is in fact aligned, contradiction.

Hence, by Lemma 27, the test for the last letter of B_{j-1} is either protected or misaligned. Using the same accounting as in Lemma 35 we conclude that we spent at most $\mathcal{O}(n)$ time on all periodic tests. □

Proof of Lemma 26 It is left to show that indeed we do not need to take into the account the time spent on comparing $S(X)$ with $S(X)$ on the other side of the equation.

Proof (proof of Lemma 26) Recall that we only test solutions of the form $S(X) = a^k$. Since we make the comparisons from left to the right in both $S(\mathcal{A}_\ell)$ and $S(\mathcal{B}_\ell)$ then

when we begin comparing letters from one $S(X)$ with the other $S(X)$, we in fact compare some suffix a^ℓ of a^k with a^k. Then we can skip those a^ℓ letters in $\mathcal{O}(1)$ time. Consider the previous test, which needs to include at least one explicit letter. Whatever type of test it was or whatever group of tests it was in, some operations were performed and this took $\Omega(1)$ time. So we associate the cost of comparing $S(X)$ with $S(X)$ to the previous test, increasing the running time by at most a multiplicative constant. □

Open Problems

- Is it possible to remove the usage of range minimum queries from the algorithm without increasing the running time?
- Can the recompression approach be used to speed up the algorithms for the two variable word equations?
- Can one use recompression approach also to improve upper bound on the number of solutions of an equation with a single variable (currently it is $\mathcal{O}(\log \#_X)$?

Acknowledgments I would like to thank A. Okhotin for his remarks about ingenuity of Plandowski's result, which somehow stayed in my memory; P. Gawrychowski for initiating my interest in compressed membership problems and compressed pattern matching, exploring which eventually led to this work as well as for pointing to relevant literature [14,16]; J. Karhumäki, for his explicit question, whether the techniques of local recompression can be applied to the word equations; last not least, W. Plandowski for his numerous comments and suggestions on the recompression applied to word equations. This work was supported by Alexander von Humboldt Foundation.

References

1. Berkman, O., Vishkin, U.: Recursive star-tree parallel data structure. SIAM J. Comput. **22**(2), 221–242 (1993). doi:10.1137/0222017
2. Charatonik, W., Pacholski, L.: Word equations with two variables. In: Abdulrab, H., Pécuchet, J.P. (eds.) IWWERT, LNCS, vol. 677, pp. 43–56. Springer, Berlin (1991). doi:10.1007/3-540-56730-5_30
3. Dąbrowski, R., Plandowski, W.: Solving two-variable word equations. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP, LNCS, vol. 3142, pp. 408–419. Springer, Berlin (2004). doi:10.1007/978-3-540-27836-8_36
4. Dąbrowski, R., Plandowski, W.: On word equations in one variable. Algorithmica **60**(4), 819–828 (2011). doi:10.1007/s00453-009-9375-3
5. Jeż, A.: Faster fully compressed pattern matching by recompression. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP (1), LNCS, vol. 7391, pp. 533–544. Springer, Berlin (2012). doi:10.1007/978-3-642-31594-7_45. Full version accepted for publication in Transactions on Algorithms. doi:10.1145/2631920
6. Jeż, A.: Approximation of grammar-based compression via recompression. In: Fischer, J., Sanders, P. (eds.) CPM, LNCS, vol. 7922, pp. 165–176. Springer, Berlin (2013). doi:10.1007/978-3-642-38905-4_17. Full version at http://arxiv.org/abs/1301.5842
7. Jeż, A.: Recompression: a simple and powerful technique for word equations. In: Portier, N., Wilke, T. (eds.) STACS, LIPIcs, vol. 20, pp. 233–244. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2013). doi:10.4230/LIPIcs.STACS.2013.233
8. Jeż, A.: The complexity of compressed membership problems for finite automata. Theory of Comput. Syst. (2014). doi:10.1007/s00224-013-9443-6

9. Jeż, A.: Context unification is in PSPACE. In: ICALP, LNCS, vol. 8573, pp. 244–255. Springer, Berlin (2014). Full version at http://arxiv.org/abs/1310.4367
10. Jeż, A., Lohrey, M.: Approximation of smallest linear tree grammar. In: Mayr, E.W., Portier, N. (eds.) STACS 2014, LIPIcs, vol. 25, pp. 445–457. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2014)
11. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. JACM **53**(6), 918–936 (2006). doi:10.1145/1217856.1217858
12. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) CPM, LNCS, vol. 2089, pp. 181–192. Springer, Berlin (2001). doi:10.1007/3-540-48194-X_17
13. Laine, M., Plandowski, W.: Word equations with one unknown. Int. J. Found. Comput. Sci. **22**(2), 345–375 (2011). doi:10.1142/S0129054111008088
14. Lohrey, M., Mathissen, C.: Compressed membership in automata with compressed labels. In: Kulikov, A.S., Vereshchagin, N.K. (eds.) CSR, LNCS, vol. 6651, pp. 275–288. Springer, Berlin (2011). doi:10.1007/978-3-642-20712-9_21
15. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Matematicheskii Sbornik **2**(103), 147–236 (1977) (in Russian)
16. Mehlhorn, K., Sundar, R., Uhrig, C.: Maintaining dynamic sequences under equality tests in polylogarithmic time. Algorithmica **17**(2), 183–198 (1997). doi:10.1007/BF02522825
17. Obono, S.E., Goralcik, P., Maksimenko, M.N.: Efficient solving of the word equations in one variable. In: Prívara, I., Rovan, B., Ruzicka, P. (eds.) MFCS, LNCS, vol. 841, pp. 336–341. Springer, Berlin (1994). doi:10.1007/3-540-58338-6_80
18. Plandowski, W.: Satisfiability of word equations with constants is in NEXPTIME. In: STOC, pp. 721–725. ACM (1999). doi:10.1145/301250.301443
19. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. JACM **51**(3), 483–496 (2004). doi:10.1145/990308.990312
20. Plandowski, W.: An efficient algorithm for solving word equations. In: Kleinberg, J.M. (ed.) STOC, pp. 467–476. ACM (2006). doi:10.1145/1132516.1132584
21. Plandowski, W., Rytter, W.: Application of Lempel–Ziv encodings to the solution of word equations. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP, LNCS, vol. 1443, pp. 731–742. Springer, Berlin (1998). doi:10.1007/BFb0055097
22. Sakamoto, H.: A fully linear-time approximation algorithm for grammar-based compression. J. Discrete Algorithms **3**(2–4), 416–430 (2005). doi:10.1016/j.jda.2004.08.016